DISTRIBUTED OBJECT SYSTEM ENGINEERING FOR TERMINAL AERODROME FORECAST VALIDATION AND METRICS PROCESSING

THESIS

James S. Douglas, Captain, USAF

AFIT/GCS/ENG/00M-07

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/00M-07

DISTRIBUTED OBJECT SYSTEM ENGINEERING FOR TERMINAL AERODROME FORECAST VALIDATION AND METRICS PROCESSING

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Systems

James S. Douglas, B.S.

Captain, USAF

March, 2000

AFIT/GCS/ENG/00M-07

DISTRIBUTED OBJECT SYSTEM ENGINEERING FOR TERMINAL AERODROME FORECAST VALIDATION AND METRICS PROCESSING

THESIS

James S. Douglas, B.S.

Captain, USAF

Approved:

[illegible]	3 March '00
Gary B. Lamont, PhD (Chairman)	**Date**
Thomas C. Hartrum	3 March 00
Thomas C. Hartrum, PhD	**Date**
[illegible]	3 Mar 2000
Major Michael L. Talbert, PhD	**Date**

Acknowledgements

I thank my wife Lynn and our children Joshua, Jonathan, and Melissa for all their support and understanding during the past 18 months - without it, the days would have been much longer and less fulfilling. Although I always tried to be there for the kid's basketball, baseball, soccer, scouting, and homework (just to name a few), it was Lynn who held the family together over the past 18 months and provided me a wonderful learning atmosphere - thank you.

To Dr. Gary Lamont for his guidance on distributed system development and patience in steering me through the thesis and scientific experimentation process. My goals when I chose to pursue a graduate program in computer science were never about making the highest grade, they were always about being able to solve problems through individual effort and research. Although I rarely (if ever) asked a professor for advice; opting to make, correct, and pay for my mistakes along the way, Dr. Lamont always seemed to provide very timely insight into issues that seemed puzzling to me at the time. I thank him for helping me to think "out of the box," and acknowledge that his influence will greatly impact my future work in the field.

To Dr. Tom Hartrum and Major Mike Talbert for their instructive feedback, constructive criticism, and sense of humor over the last 18 months. Even though I may not have absorbed everything you folks taught me, I always enjoyed your perspectives, whether they concerned object-oriented software engineering or relational databases -- I greatly enjoyed watching you folks teach. I think it finally came together for me here in the end.

Last, but certainly not least -- I'd like to thank Captain Darryl N. Leon. Darryl is a Florida State University alumni (I won't hold it against him) and an avid sports fan. Our spirited discussions on college and professional sports were as important to me as any class I was taking at the time. Darryl is also an experienced Air Force weather guy. Besides the spirited sports discussions, his weather expertise proved instrumental in helping me understand terminal forecasts and the problem I was really trying to solve – thanks for the insight.

James S. Douglas

Table of Contents

Page

List of Figures

Page

List of Tables

AFIT/GCS/ENG/00M-07

Abstract

Distributed object systems are a very complex intertwining of heterogeneous hardware, software, and operating systems coupled with communication networks of varying protocols and capacities. Distributed components offer improved performance through parallel processing, improved expansion and scalability opportunities through modularity, improved availability through replication, and improved resource sharing and interoperability through interconnection. This research provides a distributed system design methodology to validate terminal forecasts and gather metrics for the Air Force Weather Agency. Proven principles such as component reuse and architectural development are applied through the use of parameterized types and design patterns. A client/server measurement model is developed to show the impact of design decisions on computer resource utilization, system scalability, system performance, ease of implementation, and system evolution. An experimental Common Object Request Broker Architecture (CORBA) application is implemented to quantify the approach's effectiveness toward selecting an appropriate CORBA implementation and deploying the application in a distributed environment. While this research specifically uses CORBA for system development, the methodology presented is easily mapped onto any client/server architecture.

DISTRIBUTED OBJECT SYSTEM ENGINEERING FOR TERMINAL AERODROME FORECAST VALIDATION AND METRICS PROCESSING

I. Introduction

Software collaborating across machine and network boundaries to create client/server solutions is the central concept behind distributed object systems [8]. In distributed object systems, objects are typically divided between three or more tiers: a user interface layer, at least one business layer[1] that executes application business logic; and one or more data layers that provide storage and retrieval services for user data. Figure 1 shows a typical topology consisting of four layers: 1) system interface layer, 2) business layer, and 3) a data layer further subdivided into two distinct layers: a data object layer and data storage layer. Because of this organization, distributed object systems are also referred to as N-tier models. The terms *distributed systems, distributed objects, component-based systems,* and *N-tier systems* are interchangeable in this discussion.

Distributed object system technology provides several advantages over monolithic applications. Large component-based applications may be developed in small steps, with new system functionality added as a normal course of evolution. Treating distributed components as binary black boxes, instead of focusing on source code, greatly enhances component autonomy and reuse [13]. Distrib-

[1] Business layer refers to the software rules and logic required to perform application processing requirements.

uted components also offer improved performance through parallel processing, improved expansion and scalability opportunities through modularity, improved availability through replication, and improved resource sharing and interoperability through interconnection [17 and 57].

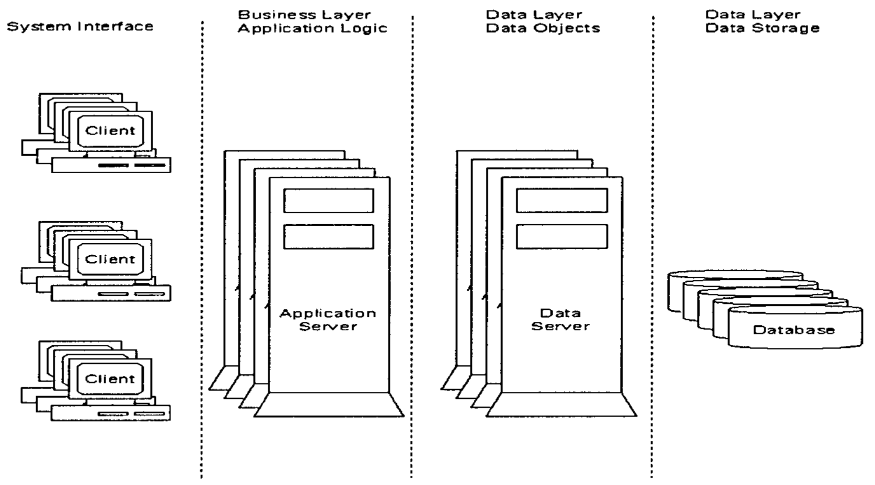


Figure 1: A Typical N-Tier Topology

This investigation relates to open system development in large heterogeneous environments. In general, this effort addresses distributed object system development using the Common Object Request Broker Architecture (CORBA) [21 and 58]. Specifically, this effort addresses distributed object system development as applied to the Air Force Weather Agency (AFWA) application domain and concentrates its efforts on Terminal Aerodrome Forecast (TAF) validation, publication, and metrics processing problems [1 and 59]. Issues such as network and resource utilization, system scalability, system performance, ease of implementation, and system evolution are discussed in great detail.

This chapter provides a background on the core problem this research addresses: **Designing an open, distributed system that validates TAF information, provides validated TAF information to AFWA aviation customers, and processes TAF accuracy metrics with no manual intervention.** This chapter also addresses specific research objectives; the significance and impact anticipated from conducting this type of research; applicable assumptions, research scope and constraints as they relate to the problem domain; and the approach and organization applied to this particular problem.

1.1 Application Domain

AFWA currently uses a manpower-intensive process to produce TAF accuracy metrics. The current process begins when a weather forecaster submits a TAF for distribution into the system. The TAF is examined for errors at AFWA. TAF accuracy is *manually* tracked on hand-written forms, then these forms are transcribed into a database. On a regular basis, this data is *collected* at the command level for report-generation. Besides the apparent process inefficiencies from duplicate data entry, there are two other inherent problems. Although a given metric is pre-defined, forecasters can track only very specific and simple metrics due to current system limitations. TAF validation, publication, and metric reporting facilities are not integrated in the current system. The terms *TAF, terminal forecast, and forecast* are interchangeable in this discussion.

AFWA/XP is currently tasked with the responsibility for enabling automated metric facilities and is therefore sponsoring research into methods that would achieve solutions to these problems. The Electronic Systems Center, Air Force Weather Systems (ESC/ACW) drafted a System Requirements Document (SRD) [2] to help AFW achieve business solutions to a wide array of information system and data processing problems [1]. The SRD outlines the specific hardware, software, and interoperability standards that ESC/ACW believes will evolve current AFW information systems toward enabling technologies like CORBA, the distributed object platform addressed and implemented in this effort.

1.2 Problem Discussion

A major reason for the inefficient TAF process is current forecast entry computer systems cannot communicate directly with backend systems used to store and track observation trends for TAF metrics collection. Therefore, a globally integrated open architecture should be at the heart of the AFW reengineering effort [1]. For users to have confidence in the collected metrics though, the issue of forecast *confidence* using data validation techniques must also be addressed. Currently, no automated *front-end* quality control (QC) is conducted on any AFW weather product. The Automated Weather Distribution System (AWDS), which currently produces weather reports at the Weather Flight (WF), has no QC capa-

[2] The ESC/AFW Systems Requirement Document is a living document outlining AFW information system evolution.

bility. This means that forecasts with typographical errors are submitted into the global network.

TAF errors are currently trapped at AFWA, so every TAF (correct or incorrect) is transmitted to AFWA for validation. A problem with this type of *back-end* QC develops when AFWA has to process many errors, usually during weather events when forecasters are extremely busy. During these time periods, AFWA *rejects* a higher number of forecasts and these rejected forecasts are not included in TAF accuracy metrics. This leads to the TAF accuracy metric reporting better-than-actual results for the time period with high rejection rates. For automated metric collection to become feasible, front-end forecast validation becomes a critical component in the overall distributed system design.

The distributed object system must integrate QC and provide validation facilities for TAF submissions, alerting the weather observer about data discrepancies *before* TAF transmission by comparing the submitted TAF to site observation trend data. The distributed object system must also provide collection facilities to report TAF/observation format and accuracy metrics on submitted weather data. This research addresses the issues of data validation and collection as they apply to the TFMS, and distributed systems in general.

1.2.1 Current Forecast Generation [46]

Figure 2 presents a high-level view of the TAF generation process. When a forecaster produces a TAF, typically another person checks the TAF for errors.

No automatic QC is performed. When the forecaster sends the TAF, the local database is updated. Typically, forecasts and observations are stored for 24 hours. The original idea was to store the forecast for validation. In practice, forecasts are rarely re-examined unless errors are reported [46].

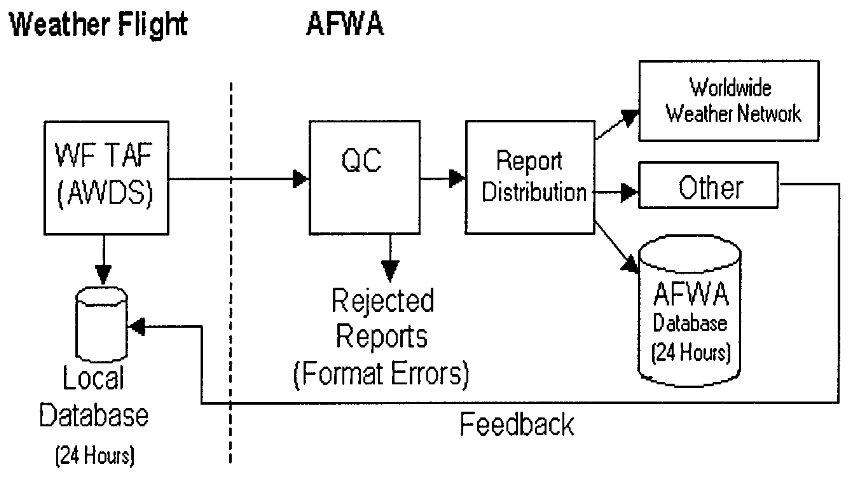


Figure 2: Current TAF generation [46]

As shown in Figure 2, errors are currently trapped at AFWA. If errors are detected, the offending TAF is rejected. Once the TAF is processed by AFWA, a copy is returned to the WF, and stored for future reference.

1.2.2 Architectural Discussion [1]

Currently, AFW is undergoing a comprehensive system restructuring. Under the new design, forecast responsibility shifts from an individual WF to an Operational Weather Squadron (OWS) servicing a particular geographic region. Because forecast responsibility is now an OWS function, TAF data is entered into the distributed system at an OWS. The OWS originates the processing and storage of local weather model and forecast data, serving as regional weather processing centers. The OWS also has the capability to initiate transactions and other coordination activities with AFWA and another OWS. OWS database systems have the capability to perform both high-speed loading of data sent from AFWA, and immediate update of individual data items sent via a DBMS synchronizer or replication facility.

AFWA basically serves a data warehouse function in the system. AFWA systems have the capability to transmit, in near real-time, new observation data to an OWS, and conversely, an OWS can transfer new observation data to AFWA. Network connectivity *within* the AFWA, OWS, Weather Flight (WF), and Detachment levels are typically LAN-grade Ethernet topologies while long-haul connectivity *between* these levels is T1-grade (1.544Mbps) or T3-grade (45Mbps) X.25 packet switching topologies.

1.3 Research Goals

First, this effort unifies traditional software engineering techniques, e.g. structured and object-oriented, to effectively develop a suitable design methodology for large-scale distributed software systems [4, 5, 6, 16, and 17]. Second, principles such as component reuse and architectural development through the use of design patterns are thoroughly investigated and applied where appropriate [13 and 14]. Third, this research builds upon previous AFIT efforts in distributed object computing [2]. Finally, this effort provides AFW with a thoroughly researched document describing 1) a methodology to design an evolutionary distributed system, 2) an experimental CORBA application that addresses TAF validation and publication distributed issues, and 3) a methodology to select an appropriate CORBA implementation by using performance benchmarks.

1.4 Specific Objectives

The principal goal of this investigation is providing AFW an appropriate distributed system design in terms of cost/performance for the given environment in which the system must operate and for the given problem (§1.2 Problem Discussion, §1.3 Research Goals). Specifically, the characteristics of and the considerations for adopting a component-based architecture within an N-tier distributed system environment are addressed. Specific objectives (research milestones) to achieve this principal goal:

1. Investigate distributed object systems – principles, benefits, and limitations.
2. Investigate distributed data systems -- architectures, protocols, and transactions. Determine design trade-offs.
3. Investigate CORBA application development.
4. Determine forecast validation and metric processing data, algorithm, and distributed characteristics.
5. Determine a suitable system concept model by analyzing the TAF problem requirements.

6. Develop a design methodology for the TFMS based on the requirements determined in step 5.
7. Using the methodology developed in step six, design a component-based distributed environment for the problem evaluated in step five.
8. Design CORBA performance benchmarks and experiments to test the environment developed in step seven.
9. Implement the environment and run experiments.
10. Evaluate using the designed measurement criteria from step eight.

1.5 Research Significance

This investigation provides a design methodology for developing large, heterogeneous distributed object systems. While this discussion specifically addresses CORBA system development, the methodology is easily mapped onto any distributed architecture, e.g. DCOM [18]. Software validation issues are addressed during system design, contributing to overall design correctness. Along with providing a sound distributed system development methodology, this research also furthers AFIT's parallel and distributed object system research by addressing CORBA system development, experimentation, and performance benchmarking.

1.6 Assumptions, Scope, and Constraints

As pointed out previously, the SRD is a living document designed to help AFW achieve business solutions to a wide array of information system and data processing problems. This effort focuses *solely* on investigating AFW TAF validation, distribution, and metrics processing in an open, distributed environment. The ESC/ACW baseline architecture is a constraint for forecast validation and metrics processing design.

An essential AFW requirement is to use technologies that permit incremental system evolution without system-wide consequences -- in other words, AFW requires a scalable architecture. The AFW architecture uses component-based and N-tier client/server technologies to achieve the desired enterprise scalability. Accordingly, the AFW system will be implemented as a set of CORBA components that interoperate through shared interfaces. The AFW architecture is an integration of both general purpose and TAF problem-specific components and needs to also support various legacy databases during the evolution to an open, component-based architecture. The SRD also specifies that hardware installations are satisfied with commercial off-the-shelf (COTS) system components. Most of the systems that ESC develops fall into this category. Software installations are hosted by a UNIX or NT platform.

1.7 Summary

This chapter provides a general description of the problem being investigated by this research effort. Research goals, objectives, significance, and impact are also discussed in great detail. Assumptions and constraints are outlined as they apply to this particular problem.

The rest of the report is organized as follows: Chapter two provides background discussions on technical subjects such as CORBA and distributed database systems. Chapters three through five addresses design methodology, measurement methodology, and implementation for an *experimental* Terminal Forecast Management System (TFMS). The TFMS is a prototype CORBA application developed throughout this investigation to collect data on distributed design issues; i.e. the TFMS is *not* a *production*-quality software system. Chapters six and seven concern gathering experimental data, analyzing experimental data, and drawing appropriate conclusions based on these empirical experimental results.

II. Distributed Object Systems

If we knew what it was we were doing, it would not be called research, would it? -- Albert Einstein

2.1 Introduction

This chapter discusses the theoretical background material applicable toward understanding the issues and complexities involved with distributed object and database systems. The typical distributed object environment is introduced to show some of the issues these systems must address and overcome when heterogeneous computer systems must communicate. Distributed object paradigms, distributed database systems, and distributed object management and performance issues are then presented and discussed.

2.2 The Distributed Object Environment

Distributed object system environments are a very complex intertwining of heterogeneous hardware, software, and operating system platforms coupled with communication networks of varying protocols and capacities. Figure 3 shows an architecture that distributed system engineers would typically encounter when developing distributed object applications.

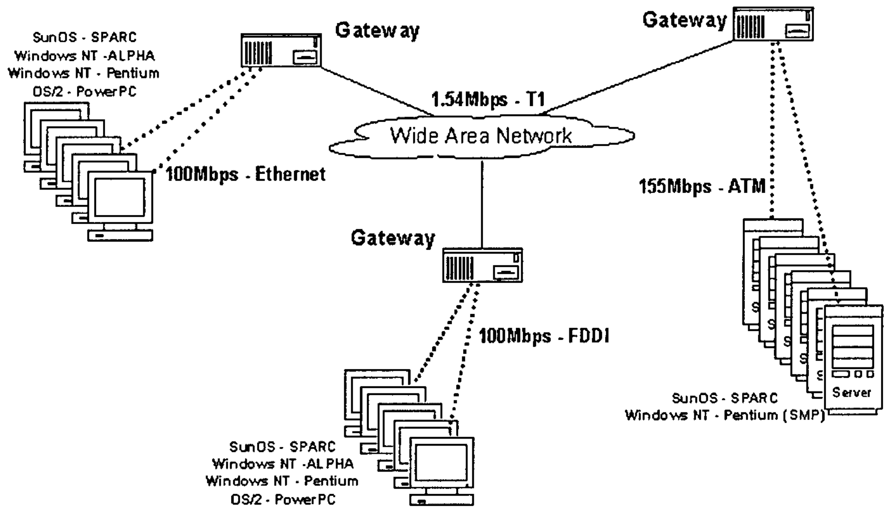


Figure 3: A Distributed Object System Environment

As shown in Figure 3, a client can operate using a number of different machine architectures, e.g. Pentium or SPARC, while running completely different operating systems, e.g. Windows NT or SunOS. Servers operate using heterogeneous hardware and software as well, but usually contain more powerful or symmetrical processing capabilities coupled with increased disk and memory capacities to handle many client requests in a completely decoupled, shared-nothing architecture (§2.5.1 Architectures). Different hardware platforms possess different instruction set architectures (ISA) with either little endian or big endian byte ordering formats; e.g. Intel uses little endian while SPARC uses big endian [12 and 55]. To shield the application developer from these portability issues, distributed application development environments provide tools such as Interface Definition Language (IDL) compilers to generate the necessary code for parameter marshaling and demarshaling [34]. Different operating system platforms possess different thread models, user interfaces (Text/GUI), and system call interfaces (Win32/POSIX) that present many issues for porting distributed applications from one platform to another [21 and 50].

A heterogeneous communications network of LANs, MANs, and WANs (Ethernet, FDDI, ATM, and T1) connects clients and servers. The network communications protocols used may be based on any of the WAN or LAN protocol families such as TCP/IP, IPX/SPX (Novell), and X.25. All these protocol families provide end-to-end communication, but they also possess different characteristics that complicate software interoperability [60]. Network characteristics will greatly affect the communication time in the distributed object system. On the LAN side, network device (hubs/switches) characteristics such as shared or switched bandwidth greatly affect communication time. On the WAN side, network bandwidth (100Mbps LAN versus 1.54Mbps WAN), communication latency, delays caused by buffering interconnection device mismatches, e.g. gateway connections from high-speed LANs to the WAN, and fragmentation/reassembly operations between different protocol families all greatly affect distributed application performance [10 and 11].

With all of these complexities, distributed object systems attempt to address the following issues:

1. Location Transparency: The client does not know what server hosts a particular target object. The server/object is free to move to any location [21].

2. Language Independence: The client does not care what implementation language the server uses. The language can change without affecting clients.
3. Implementation Independence: The client treats the target object as a black box, i.e. the client possesses no knowledge of the target object's implementation. In distributed object software engineering, it's absolutely critical to program to an interface, not an implementation [12].
4. Architecture Independence: Client is unaware of the server's underlying hardware architecture or ISA.
5. Operating System Independence: Client is unaware of the server's operating system, use of threads or event loops.
6. Communication Independence: Client is unaware of the transport and protocols used for method invocation. ORBs transparently connect clients to servers (§2.3.1 Common Object Request Broker Architecture (CORBA)).

2.3 Distributed Object Paradigms

Distributed object system architectures are a development evolution rooted from previous forms of software development methodologies: modular, functional, client/server, and most notably -- object-oriented. In a pure object-oriented approach, reuse and inheritance is restricted to the source code level. If a developer changes a class definition, she would have to change and recompile the entire application. The idea behind components is to promote the *binary* reuse of software. In a distributed object system, the *component* is the unit of packaging, distribution, maintenance, and development [17]. The component acts as a service provider, responding to messages sent to its published interface. Interface implementations are hidden from clients, so components may change internally, provided that their interface definition is maintained.

Since components are standalone objects that can plug-and-play across network, application, and language boundaries, they also provide fine, medium, and course grain parallelism capabilities in the distributed architecture. By simply migrating or replicating components or objects to appropriate network nodes, we provide performance, fault tolerance, and load-balancing capabilities that didn't previously exist. The *goal* of component-based development is to provide software users and developers the same level of application interoperability that is currently available to users and developers of electronic parts such as integrated circuits [8]. In this section, the two leading distributed object systems are explored: the Common Object Request Broker Architecture (CORBA) and the Distributed Component Object Model (DCOM). The Java Remote Method Invocation (RMI) and Enterprise JavaBeans models are not discussed due to their lack of language independence support.

2.3.1 Common Object Request Broker Architecture (CORBA)

To support very large, complex distributed object applications, it's desirable to specify an infrastructure that supports the handling of common operations such as object lifecycles, identification, interface definitions, and intercommunication. The Object Management Group (OMG) was formed to reduce complexity and lower development cost and time. The OMG is an international trade organization incorporated as a nonprofit organization in the United States. OMG is currently comprised of over 800 corporate members and the number gets larger every year [21].

Metadata is a crucial ingredient when developing flexible distributed systems. Metadata provides a distributed system with self-describing, dynamic, and reconfigurable capabilities. Using metadata, components discover each other at runtime, further enhancing interoperability [8]. The OMG Interface Definition Language (IDL) provides the "glue", connecting objects in a standard way by defining the interfaces to CORBA objects. Because IDL is a *declarative* language, its sole purpose is to allow object interfaces to be defined in a manner entirely independent of any particular programming language [21]. This allows applications implemented in different programming languages to interoperate; this language neutrality is critical to CORBA supporting heterogeneous environments [8, 21, and 26]. Language mappings specify how IDL is transformed into a particular programming language, e.g. in C++, interfaces transform to classes and in Java, interfaces transform to public interfaces.

Figure 4 shows the components an ORB uses to transmit requests transparently from a client to a particular object implementation (servant). When the client invokes an operation on a servant, CORBA's runtime infrastructure, the

ORB Core, delivers the request for the client. For remote servants, the ORB Core uses the Internet Inter-ORB (IIOP) communication protocol to deliver the request and return the servant's response (if any). The ORB Interface is a CORBA-compliant standard interface written in IDL that provides standard operations such as ORB initialization and shutdown.

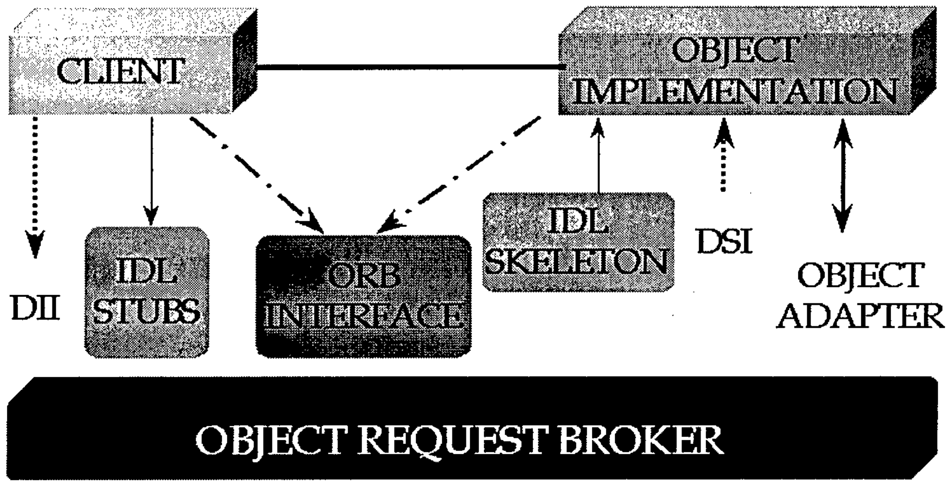


Figure 4: CORBA Specification

IDL compilers generate IDL stubs and skeletons (proxies). IDL stubs and skeletons are the "glue" in CORBA, holding clients and servants together. IDL Stubs provide strongly typed interfaces to clients that hide many low-level networking details, e.g. marshaling data into a packet-level format. IDL skeletons perform the server-side analog by unpacking the packet-level format into typed data for the application. IDL compilers provide language transparency by transforming OMG IDL definitions into implementation languages, e.g. C++ and Java. Besides providing language transparency, IDL compilers also eliminate many sources of network programming errors while also providing optimization opportunity [56].

The Dynamic Invocation Interface (DII) is an alternative to IDL stubs for clients to "discover" and invoke objects. While static stubs provide an object type-specific API, DII provides a generic mechanism for constructing requests at run time. The interface repository (not shown), a client's object definition database of metadata, allows some measure of type checking to ensure that a target object can support the request made by the client.

The Dynamic Skeleton Interface (DSI) is the server-side counterpart of DII. While IDL skeletons invoke specific operations in the object implementation, DSI defers this processing to the object implementation repository. This is useful for developing bridges and other mechanisms to support inter-ORB interoperation. The implementation repository is the server analogue to the interface repository; this is the server-side object definition database.

The Object Adapter (OA) associates a servant with an ORB, demultiplexes requests, and dispatches the appropriate operation upcall on a particular servant. The OA also provides extensibility of a CORBA-compliant ORB to integrate alternative object technologies into the OMA. For example, adapters may be developed to allow remote access to objects that are stored in an object-oriented database. Each CORBA-compliant ORB must support a specific object adapter called the basic object adapter (BOA). CORBA release 2.2 specifies the portable object adapter (POA), which removed server-side portability problems that existed in the BOA [21]. CORBA is part of the Object Management Architecture (OMA). More information on the OMA can be found in Appendix A.

2.3.2 Distributed Component Object Model (DCOM)

DCOM is Microsoft's proprietary standard that extends the COM model allowing objects to dynamically interact across a network. DCOM simply replaces the standard COM inter-process communication by a network protocol. The terms COM and DCOM are used interchangeably, but COM more adequately describes a single machine application, while DCOM describes a multi-computer (network) application. As with the OMA, COM objects expose their services using an interface defined by an IDL.

COM Components are created as executable code, distributed as Win32 dynamic link libraries (DLLs) or executables (EXEs). COM components support many object-oriented characteristics such as polymorphism, encapsulation and interface inheritance, but do not support implementation inheritance. Binary reuse is accomplished using containment and aggregation facilities. COM components are usually represented as shown in Figure 5. COM components are language independent and the library API provides the common component management services.

All COM components are registered in the Microsoft Windows registry. The registry serves as a local repository of object definitions that the client uses to lookup interfaces (paths) to objects. All interfaces have a unique identifier number called an *IID* or interface ID. Object packages have a unique identifier number called a *CLSID* or class ID. These IDs are Global Unique Identifiers (GUID), generated by an algorithm using the network board physical address, generation time, and other variables to ensure the generated ID is unique [17].

The client application uses COM objects through COM interfaces. During the first request (or at a time specified by the client), the server object is activated and the requested interface is sent back to the client. All COM interfaces are derived from a standard interface called *IUnknown*. An object can implement one or more interfaces using the containment and aggregation facilities as mentioned above. Two types of server objects exist in COM. An *in-process* object is packaged as a DLL and executes inside the client's address space while an *out-of-process* object is packaged as an EXE, capable of residing on the same machine in a different address space, or in a different machine altogether.

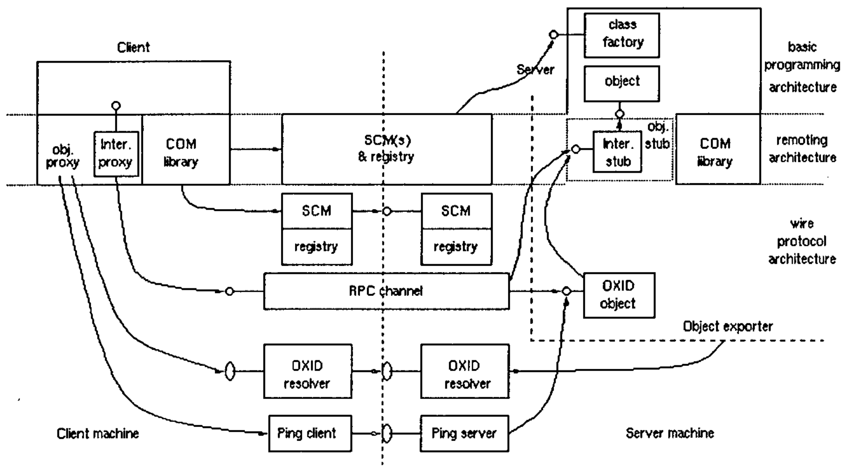


Figure 5: DCOM overall architecture [17]

Clients access server objects using a part of the COM infrastructure called a *proxy-stub* pair. The proxy-stub pair's purpose is to transfer parameters and return values across different address spaces or physical machines - a process called *marshaling* [12].

All COM calls are synchronous, so scalability becomes an issue for large applications. To circumvent this shortcoming, Microsoft provides asynchronous capabilities to COM applications by using an additional middleware platform -- Microsoft Message Queue Server (MSMQ). COM+, the ensuing COM release, will integrate MSMQ into the core COM architecture making asynchronous COM calls a built-in architectural feature for COM+ application developers.

2.4 Distributed Object Security

Distributed application development is a very complex undertaking with many considerations. Although physical limitations such as machine ISA, remote call dispatch rate, and network utilization issues require careful consideration, new issues related to security are raised as well. Clients and components need to communicate with each other securely. Since many component operations are now physically accessible by anyone with network access, this raises many issues of privacy and knowing that a given client or component is authorized to use published operations. This section looks at distributed security issues using Kerberos as a case study security service.

2.4.1 Security Issues

Distributed systems are more vulnerable to security incidents than traditional client/server applications. In a distributed application, components can't trust any other component in the distributed system to protect its resources from unauthorized access. Even if components were secure, the communication network is highly accessible to monitoring devices recording network traffic, and possibly introducing malicious code into the distributed system. The following issues highlight the difficulties in distributed object system security [8]:

1. Distributed objects may be both a client and a server: Typically, servers are trusted and clients are not trusted. Components are not easily placed in the client or server category, since a given component may perform both roles. How is a component trusted?
2. Distributed objects continually evolve: An object implementation, e.g. a mediator, may delegate some processing tasks to runtime objects; this implementation may change over time, as the component evolves. How do authorized clients know these runtime objects are secure?
3. Distributed objects are dynamic and can scale enormously: Every component is capable of becoming a server, so how are access rights managed for millions of component servers?

4. Distributed objects can be just about anything: There are probably as many different object designs as there are software designers. How do we know a component is not an imposter?

These are just a few concerns that IT professionals must address when placing their organization's valuable data on the wire. Without security support from a distributed platform, each application is forced to implement its own security mechanisms. The application would typically have to validate user credentials against a user database and return some security identifier, i.e. token, for use in future method calls. All subsequent secure method calls require the client to pass this security token. This approach works fine for a single distributed application, but how do different applications interoperate? Moving the security controls into a security service solves this problem.

2.4.2 Kerberos Authentication Service

The Kerberos protocol basically consists of three entities: a client (principal), a server, and a trusted third party to mediate between them. A *principal* is a user or process that requires secure communication [12]. Each principle is uniquely named by a principal number [74]. The trusted intermediary in this protocol is the Kerberos authentication server, a.k.a. the Key Distribution Center (KDC) or Ticket Granting Server (TGS). Kerberos *authentication* provides a means of verifying the identities of clients and servers, i.e. a principal [12]. Kerberos performs authentication using a secret key cryptosystem. A secret key cryptosystem uses a single key for both encryption and decryption; this key is called a session key in the Kerberos protocol [74]. Clients prove their identity by presenting *tickets* and *authenticators* to servers [12 and 74]. A *ticket* is an *encrypted* data structure issued by the KDC or TGS that identifies a principal and session key [74]. An *authenticator* is an *encrypted* data structure that proves a principal actually possesses the session key. The Kerberos Version 5 protocol is implemented for a variety of systems, most notably Microsoft Windows 2000, where it's used as an authentication service for distributed system security [75]. The Kerberos RFC basically addresses three security services: 1) authentication, 2) data integrity, and 3) data privacy.

Authentication [74]: The Kerberos authentication process is shown in Figure 6. The notation generally used to indicate encryption is Cyphertext = {Plaintext} Key [12]. In Figure 6, all security components are spelled out to prevent confusion. First, the client sends a request to the KDC by identifying itself, presenting a nonce (message identifier), and requesting a ticket for a given server process [12 and 74]. The KDC creates a random encryption key (session key) and generates a ticket for the requested server process. The KDC encrypts the session key and nonce using the *client's* secret key. The KDC encrypts the ticket, which consists of the session key, authorization data, principal name, Kerberos realm, and valid time period (to name a few) using the *server process'* secret key [75].

Upon receipt, the client caches the ticket, and decrypts the session key for future use. The client generates an authenticator that contains a current timestamp, then encrypts the authenticator using the session key. The client then transmits the ticket and the newly generated authenticator to the server process. The server decrypts the ticket using its secret key, and extracts the identity of the client and the session key from the ticket.

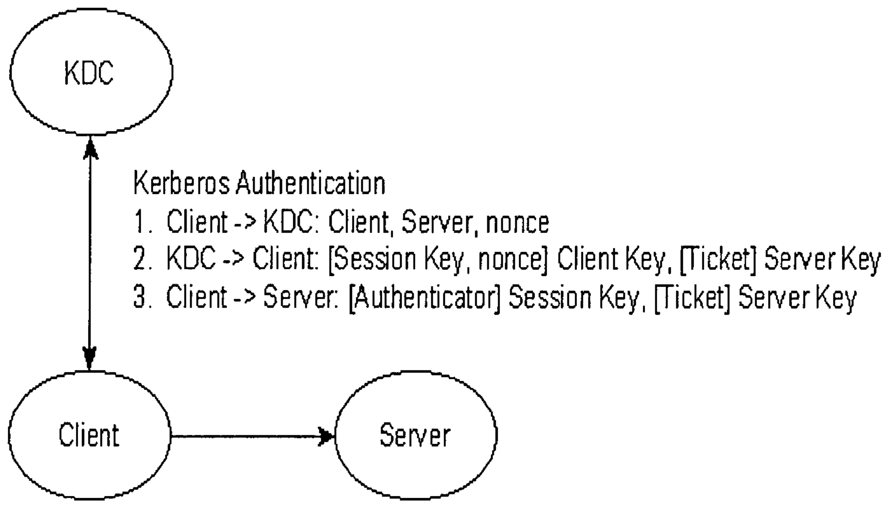

KDC
Kerberos Authentication
1. Client -> KDC: Client, Server, nonce
2. KDC -> Client: [Session Key, nonce] Client Key, [Ticket] Server Key
3. Client -> Server: [Authenticator] Session Key, [Ticket] Server Key
Client
Server

Figure 6: Kerberos Authentication

To authenticate the client, the server process decrypts the authenticator using the *extracted* session key, and verifies the timestamp is current. Successful verification of the authenticator *proves* that the client does indeed possess the session key, because the client could only obtain the session key by successfully decrypting the session key sent from the KDC with its very own key. The session key is optionally used to authenticate the server process (mutual authentication) by requiring the server process to send a fresh message encrypted using the session key to prove its identity to the client. It may also be used to encrypt further communication between the two parties or to exchange a separate sub-session key to be used to encrypt further communication.

The Kerberos protocol is also designed to operate across organizational boundaries. A client in one organization can be authenticated to a server process in another organization. Each Kerberos organization establishes a *realm* and authentication is performed using an inter-realm key [74]. The exchange of inter-realm keys registers the ticket-granting service of each realm as a principal in the other realm. A client is then able to obtain a ticket-granting ticket for the remote realm's ticket-granting service from its local realm. Tickets issued by the remote ticket-granting service will indicate to the server process that the client was authenticated from another realm.

Data Integrity and Data Privacy: Message integrity between principals can also be guaranteed using the session key. This approach provides replay attack detection and message stream modification attacks. It is accomplished by generating and transmitting a collision-proof checksum, i.e. a hash or digest function, of the client's message, keyed with the session key [75]. Privacy and integrity of the messages exchanged between principals can be secured by encrypting the data to be passed using the session key passed in the ticket.

2.5 Distributed Object Storage

Database systems play an important role in distributed object system design. At some point in a distributed architecture, data will be persistently stored.

The way data is created, read, or updated can greatly impact performance and overall distributed system design. This section provides a brief discussion on the major areas designers need to consider.

2.5.1 Architectures

There are three typical architectures used for parallel and distributed database systems as shown in Figure 7 [20]. Parallel database *machines* typically use the shared-everything or shared-disk layouts as the prominent feature is one of tight hardware coupling while a distributed database typically employs the shared-nothing architecture, as the nodes are loosely coupled or physically separated.

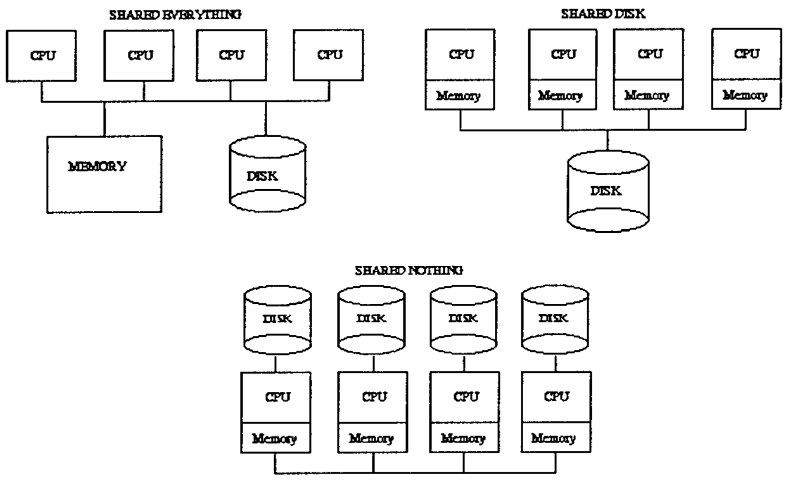


Figure 7: Parallel and Distributed Database Architectures

The most common architecture for small to mid-sized parallel database machines is *symmetric multiprocessing (SMP),* commonly called shared-everything where multiple processors are used in parallel. The most important drawback to this architecture is the von Neumann bottleneck: CPU to memory bus contention.

The opposite of shared-everything is shared-nothing, which has been the architecture of choice for designers of *highly scalable* parallel database systems [20]. In this system, each node is independent, with its own CPU(s), memory, and disk. This approach eliminates the von Neumann bottleneck, but it requires complex data fragmentation and allocation schemas in order to achieve optimal performance [28].

Combining these two extreme architectures into a design that takes advantages from both have resulted in the shared-disk architecture, somewhat similar to shared-nothing, except centralized disk farms are used for storing data. This approach eliminates almost all data partitioning problems associated with the shared-nothing architecture, but possesses the scalability problems associated with contention for a centralized resource – in this case, the disk farm [12 and 20]. New clustering products such as Microsoft Cluster Server for Windows NT have "reincarnated" shared-disk architectures, which currently found a market niche providing failover services for mission critical business applications [35].

2.5.2 Transactions

Transactions have four properties (commonly referred to as ACID properties) [9]:

1. Atomicity: Transactions are atomic (all or nothing).

2. Consistency: Transactions transform the database from one consistent (known) state into another consistent state.
3. Isolation: Even though there are many transactions running concurrently, updates are concealed from other transactions until a *commit* occurs.
4. Durability: Once a transaction commits, its updates are never lost.

Different sorts of parallelism that one can consider for distributed transactions:

1. *Task spreading:* A task is divided into a number of similar subtasks, which execute on different nodes. This requires a coordinating process that hands out subtasks, and then collects and combines the results.
2. *Pipeline:* A task is decomposed into subtasks, which execute consecutively in the same datastream [38]. Subtasks are assigned to different processes, each receiving input from a predecessor subtask, and each sending a result to a successor subtask. Every subtask is activated upon receiving its first data. If the last subtask gets its first input *before* the first subtask sends its last output, we will have *all* processors working in parallel.

Task spreading and pipelining techniques are used in different ways for distributed transaction or query processing:

1. *Intra-operator parallelism* lends itself to pipelining since *one* operation in a query tree is distributed over more than one processor (same data).

2. *Inter-operator parallelism* lends itself to task spreading since *many* operations in a query tree may be executed on different nodes concurrently, a.k.a. independent parallelism [38].

2.5.3 Serializability

Centralized synchronization mechanisms such as the two-phase commit protocol (§2.5.4 Recovery Protocols) are extensions of centralized control methodologies applied to a distributed environment where the notion of *serializability* is the generally accepted criterion for correctness. Serializability refers to a set of interleaved transactions, which produce the same result as executing the transactions individually, i.e. serial and interleaved schedules are equivalent [9].

The basic serializability concept is the same for a distributed environment, but with the added complexity imposed by transaction distribution. For distributed transactions, we require local serializability of all local schedules *and* global serializability for all global schedules, i.e. all sub-transactions of global transactions appear in the same order in the equivalent serial schedule at all sites [28]. For example, if we have two *global* transactions T1 and T2, each having two sub-transactions at *sites* A and B, then we can say T (A1) and T (B1) are sub-transactions of global transaction T1; and T (A2) and T (B2) are sub-transactions of global transaction T2. Distributed serializability means we have local ordering at sites A and B where T (A1) < T (A2) and T (B1) < T (B2) and global ordering for transactions T1 and T2 where T1 < T2 for sites A and B.

2.5.4 Recovery Protocols

An important goal in distributed data systems is that a failure of one site should not affect the processing in another site, i.e. operational sites should not be left *blocked,* waiting on a failed or otherwise unresponsive site [9 and 28]. In this section, two common protocols suitable for use in a distributed environment are discussed: two-phase commit and three-phase commit. The following discussion assumes a global or Distributed Transaction Coordinator (DTC) exists and the DTC possesses knowledge of transaction *agents.* Transaction agents (a.k.a. *brokers* or *transaction monitors*) are the other sites participating in the transaction.

The two-phase commit protocol ensures that the outcome of a transaction is consistent across all transaction managers (TM) participating in the transaction. As the name suggests, the protocol operates in two distinct phases to ultimately commit or abort a transaction. Figure 8 illustrates the two-phase operation for a *commit.* The two-phase commit is a *blocking* protocol. Phase one evaluates the condition of each resource manager (RM) to perform the transaction. The DTC communicates with each *local* TM to determine if the local RM is prepared to commit the transaction. For the local TM to return a *prepared* message, the local RM must guarantee that it can commit the update i.e. the RM must force-write all log entries for local resources used by the global transaction [9]. Each *local* TM responds to the DTC that it can or cannot guarantee its results.

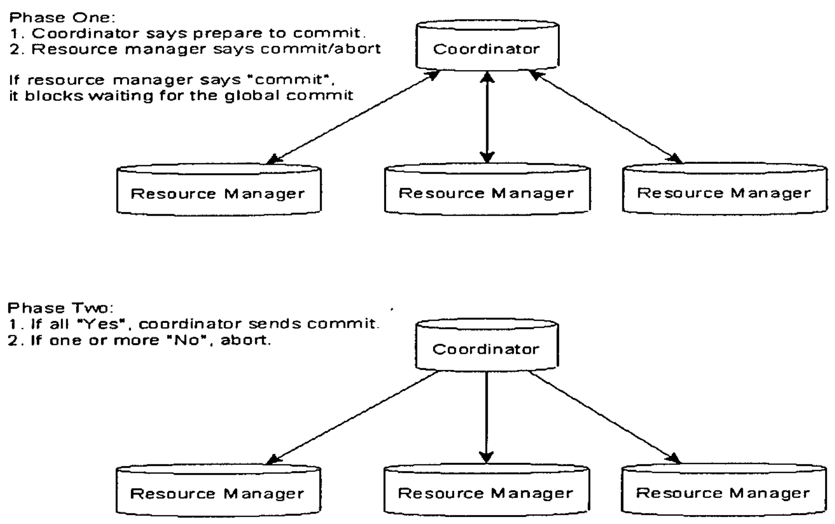

Phase One:
1. Coordinator says prepare to commit.
2. Resource manager says commit/abort
If resource manager says "commit", it blocks waiting for the global commit
Coordinator
Resource Manager
Resource Manager
Resource Manager
Phase Two:
1. If all "Yes", coordinator sends commit.
2. If one or more "No", abort.
Coordinator
Resource Manager
Resource Manager
Resource Manager

Figure 8: Two-Phase Commit Diagram

Phase two completes the transaction. When the DTC receives replies from each local TM, it force-writes an entry to its own physical log, recording its decision regarding the transaction [9]. If all participants agree, the decision is commit. If at least one participant does not agree, the decision is abort. In either case, the global coordinator informs all participants of the decision and *all participants must commit or rollback the transaction as instructed.*

If an *agent* site should fail during the transaction, its restart procedure will look for the decision record in the DTC's log. If the site finds a commit decision, then the local TM can perform a *forward* recovery by redoing previously written transaction work. If it finds an abort or no decision, then the local TM can perform a *backward* recovery by undoing state changes to restore the local database to its state prior to the transaction. If the DTC fails, a *termination* protocol must be invoked. The simplest termination protocol is for the participants to wait for the DTC to come back up or time out, whichever is sooner. A more realistic approach known as the cooperative termination protocol modifies the standard two-phase algorithm. The cooperative termination protocol requires the DTC to send a list of transaction participants to all agents. If the DTC fails, agents can contact other agents to determine if the DTC made a decision to complete the transaction. If the DTC failed before making a decision, then they can elect a new coordinator [28]. Although this method reduces the chances of blocking, blocking is still possible.

An alternative *non-blocking* protocol to the two-phase commit is the three-phase commit protocol. To arrive at the three-phase protocol, a *pre-commit* phase is added to the two-phase algorithm at the point where the participants vote and the global coordinator sends its decision regarding either a global commit or abort.

In the pre-commit phase, once the DTC receives all participant commit or abort votes, it sends a global pre-commit message, which tells all participants that they will definitely commit or rollback in due course. Each participant acknowledges receipt of the pre-commit message and continues local processing. Once the coordinator has received all acknowledgements, it then issues the global commit or abort. The important point to this algorithm is that all *operational* agents are informed of the global decision *before* the global commit or abort instruction is issued, so agent sites are not blocked waiting on this global decision and therefore act independently in the event of failure [28].

2.5.5 Object-Relational Data Mapping

This section provides a background that considers the issue of architecting object-oriented applications for high performance with relational databases. Key ideas in this section are the optimization of data objects to relational table representations and data object cache management using the *mediator* design pattern [13]. While this section primarily explores the mapping between business object instances and relational tuples using the mediator pattern, the basic idea of providing a managed cache atop a RDBMS to improve performance directly relates to forecast availability and performance in the TFMS application.

Figure 9 shows an object-relational application that provides an OO interface to underlying relational data [41]. Applications retrieve and store data using the object cache interface while a cache manager manages cache contents based on a collection query predicate [43].

A well-managed object-relational cache application addresses two major components:

1. Object-Relational Mediator(s): The mediator uses an adapter design pattern to map objects to relational tuples and vice versa [13, 41, and 43]. Business objects are managed in a shared cache on behalf of several applications. The shared cache contents are established using a collection predicate, which

specifies collection membership [43]. Tuples in the RDBMS are mapped to object instances and any updates to business objects are mapped to update operations on the RDBMS.

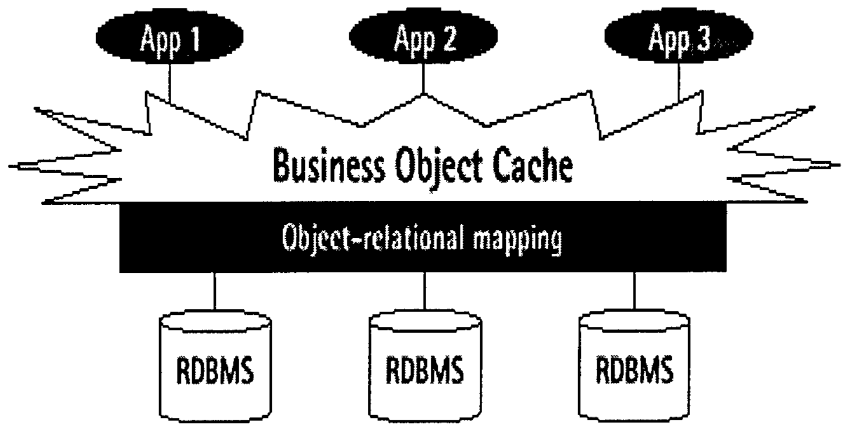


Figure 9: Object Cache [41]

2. Performance Optimizations: This component realizes that we must choose appropriate object to relational mappings, and also take advantage of relational database query and performance optimizations. Items such as stored procedures and batch operations need to be considered when designing this interface, as every advantage needs to be exploited.

As Figure 10 shows, each object instance maps to a corresponding relational table. A separate table is used for each object *type*. A simple way to map one-to-one and one-to-many relationships is to use an embedded foreign key field for each object and/or object collection relationship. The other mapping shows how a *lookup* table is employed to map many-to-many relationships. Since a cache lookup is much faster than a disk access, the goal when using an object cache is similar to operating system paging structures, where you try to maximize the *cache hit ratio* to improve performance [11].

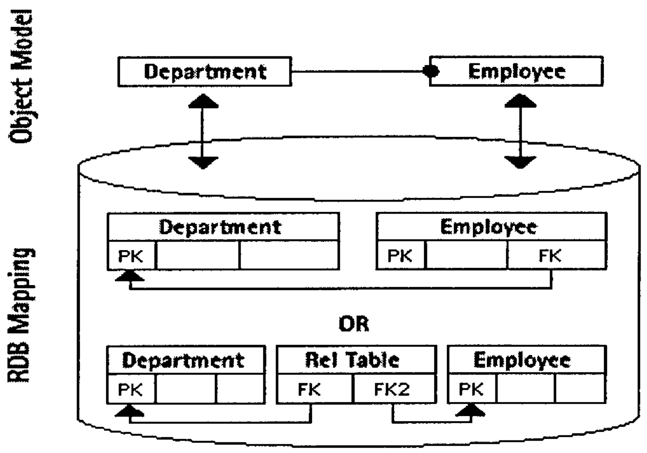


Figure 10: Object-Tuple Mapping [41]

2.6 Summary

This chapter reviews CORBA and DCOM, the two predominant distributed object paradigms. An introduction to distributed database architectures, transactions, and recovery protocols is also presented along with object-relational mapping performance issues.

III. Distributed System Design

There is no silver bullet. - F. Brooks

3.1 Introduction

The following *characteristics* have been identified as key ingredients toward developing large, distributed component software systems, and when consistently applied, produce good development results [3, 4, 5, and 6]:

1. Use Case Driven (transaction - oriented)
2. Architecturally Focused
3. Documentation-Based
4. Evolutionary Process Model: Incremental, Iterative, and Integrative

The lifecycle process used in this investigation possesses these four characteristics. The terms *use case scenario, transaction model,* and *use case* are used interchangeably throughout this section. The software development lifecycle chosen for this investigation:

1. *Requirement Analysis* captures the business model and its processes while *Analysis Modeling* specifies these requirements using different views, e.g. object/entity-relationship models, data flow diagrams, interaction/state diagrams. These two phases are combined into a *Software Requirements Specification* in this report.
2. *Design Modeling* maps the analysis model to a software architecture. In this report, this phase is referred to as *Distributed Object System Design.*

3. *Implementation Modeling* maps the software architecture to the processing environment or hardware architecture.
4. *Coding* applies development tools to the implementation model.
5. *Quality Assurance, testing,* and *experimentation* validates algorithm selection and code functionality.
6. *Evolution and Maintenance* is incremental, iterative, and integrative.

This chapter addresses software requirement specification and distributed object system design.

3.2 Motivation

Organizations that bind their business functionality to a specific technical implementation are faced with the prospect of having to continually re-engineer basic business rules as the system evolves. By using Object-Oriented Software Engineering (OOSE) and modular software construction methods, business functionality is completely *isolated* from all underlying technical constructs. When these technical constructs are re-engineered for performance reasons or extended as part of normal system evolution, the business model presented to customers remains unaffected. The simple organizational architecture depicted in Figure 11 coveys powerful OOSE/modularity principles and serves as a primary motivation toward applying a modular, OO design methodology.

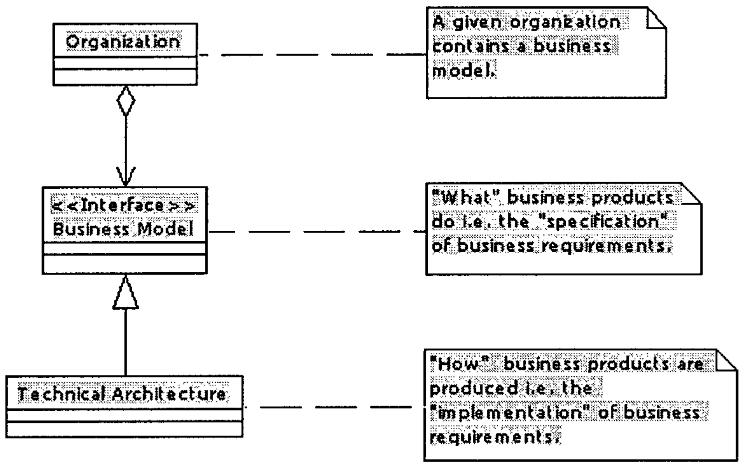

Organization
A given organization contains a business model.
<<Interface>> Business Model
"What" business products do i.e. the "specification" of business requirements.
Technical Architecture
"How" business products are produced i.e. the "implementation" of business requirements.

Figure 11: Organizational Architecture

As Figure 11 illustrates, the business model provides an interface to reflect "what" organizational services and products are advertised to consumers. The technical architecture implements the organization's business model (interface) in a particular architectural environment, so it's an extension or derived from the business model [4]. This *separation* of interface from implementation is the key to minimizing implementation dependencies and maximizing reuse in object-oriented systems [13 and 14]. In distributed object systems, this separation of interface from implementation is *strictly* enforced in IDL (§2.3 Distributed Object Paradigms). Oddly enough, many OO methodologies don't *emphasize* this key principle in developing reusable OO systems [5, 6, and 29].

3.3 Proposed Forecast Generation

Figure 12 shows a proposed AFW architecture using CORBA (§2.3.1 Common Object Request Broker Architecture (CORBA)) as the distributed object architecture. The TAF validation facility (QC) is now located at *all* OWS sites including AFWA (§1.2.1 Current Forecast Generation). Forecasts are validated upon entry into the global network and *replicated* to meet performance, availability, and AFW dissemination requirements IAW AFMAN 15-111 [67]. This architecture follows a three-tier design methodology where clients (weather customers and forecasters) form the user interface, application logic (TAF validation rules) form the middle-tier (application or web servers), and products (forecasts, observations, and statistics) form the data-tier (§5.3.1 N-Tier Topologies) [2, 4, 7, 8, 16, 27, and 32].

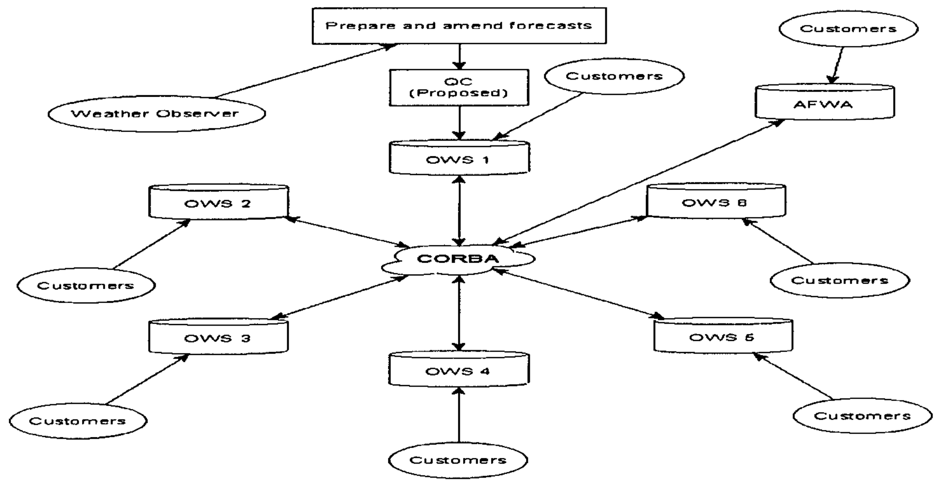


Figure 12: Proposed Architecture

Figure 13 shows a more detailed view of an OWS conforming to this design methodology. The user interface may consist of thin-client customers, e.g. web browsers, and thick-client forecasters, e.g. Win32 applications. The middle-tier consists of business logic (TAF validation rules) located on application servers. The data-tier consists of TAF products accessible either through an object cache or the underlying datastore. Many of the advantages obtained by using the N-tier approach are mentioned in the introduction, but the largest advantages concern resource scalability and configuration flexibility (§5.3.1 N-Tier Topologies).

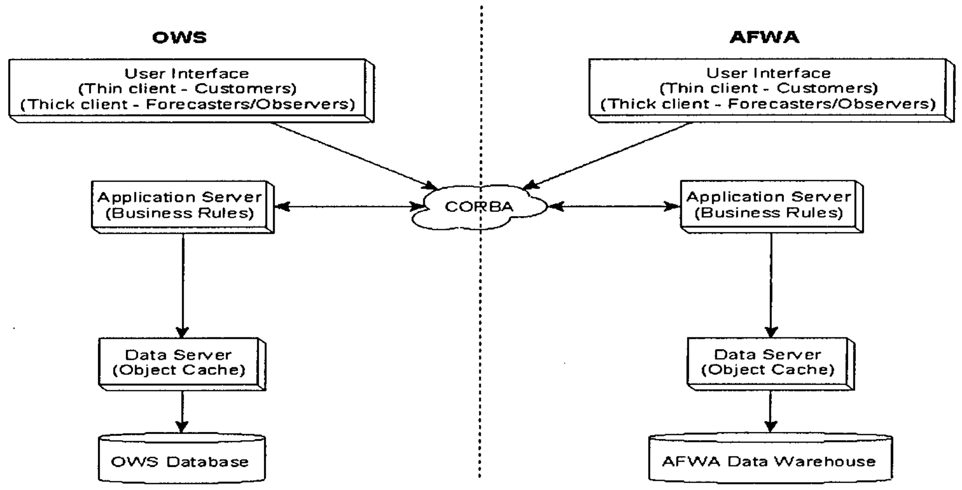

OWS
AFWA
User Interface (Thin client - Customers) (Thick client - Forecasters/Observers)
User Interface (Thin client - Customers) (Thick client - Forecasters/Observers)
Application Server (Business Rules)
CORBA
Application Server (Business Rules)
Data Server (Object Cache)
Data Server (Object Cache)
OWS Database
AFWA Data Warehouse

Figure 13: Proposed N-tier Architecture

Well-defined distributed object systems permit you to add, delete, or modify business rules (TAF validation rules) without affecting clients or any other component in the system. This is a very important consideration when developing large distributed systems because functionality may be added incrementally [13 and 14]. Monitors (agents) can observe trend data and send instructions or alarms (react) if current conditions warrant an amend action. If an event channel is used, all OWS sites could register for these alarms and receive them automatically, providing additional real-time safety updates to TAF customers at their respective sites [21 and 32].

In the proposed process, the OWS database contains forecast, observation, weather station, and statistical data. When a TAF is generated, the encoding and validation function is performed at the originating OWS, using AFW-defined encoding instructions, MAJCOM-defined weather element category thresholds, and observation trend data for the particular station [46, 59, and 68]. The TAF is then distributed by the *originating* OWS to meet near real-time performance and availability requirements [67]. This research also addresses a suitable TAF replication facility (§5.4.2 Publication Transaction) for a CORBA environment.

3.4 Software Requirements Specification

Software requirement specification captures user requirements and transforms these requirements into a detailed description of the distributed software architecture. The system's information, functions, and behavior are analyzed and the system is partitioned accordingly [6]. Software testing and validation criteria are also developed during this phase to provide *traceability* to user requirements. System constraints, performance requirements, and architectural considerations (§1.2.2 Architectural Discussion, §1.6 Assumptions, Scope, and Constraints) were previously discussed and are not repeated.

3.4.1 Business Description

A business description describes how business services are provided. Issues affecting business products, customers, and processes are typically discussed, as this description is an essential input into the Information System Model (§3.4.2 Information System Model) [27].

3.4.1.1 Overall Description

The TFMS offers *terminal* forecasting services to weather customers. A terminal forecast consists of a *concise* statement of the expected meteorological conditions significant to aviation at a specific airport during a specified time period. Terminal forecasts are prepared, issued, and distributed every six hours and are valid for a 24-hour period [59]. Amendments to a given TAF are issued as needed. These scheduling requirements are needed to satisfy the United States Aviation Authority, the Federal Aviation Administration (FAA), and the International Civil Aviation Organization (ICAO). A given TAF is coded in a format based on World Meteorological Organization (WMO) aerodrome forecast code, FM 51, and Aviation Routine Weather Report (METAR) code for both domestic and international use [38 and 59].

The TFMS validates forecasts by comparing the forecasted weather to actual weather observations, at specific times (usually hourly) during the valid period of the forecast. Forecast *verification* is the AFW term used to denote the feedback provided a weather analyst on how his forecast compares to actual weather [68]. In this report, the term *validation* is used instead of verification. In both the context of software testing and forecast verification, validation refers to the software functionality being traceable back to a specific user requirement [6].

Several statistics are computed to provide the analyst feedback on her forecasting skill.

3.4.1.2 Functional Description

The TFMS consists of the following major application functions:

1. Weather Analyst Management: This function is the client interface to the distributed system. A weather analyst may perform the following actions:
   a. Submit a new, amended, or corrected forecast.
   b. Submit a new or corrected observation.
   c. View TAF accuracy reports for a given set of weather stations, weather elements, categories, and time period; i.e. day, month, quarter, or year.
   d. View TAF, observation, and statistical data that generated the TAF accuracy report, i.e. computational source data.
2. Terminal Forecast Validation Functions: The terminal forecast validation function validates all new, amended, or corrected TAF and observation data entered into the distributed system. Major functions:
   a. Accept TAF or observation data from weather analysts.
   b. Validate all TAF and observation weather entries for encoding format errors [59 and 67].
   c. Validate all TAF weather elements using weather station trend data [46].
   d. Update station trend data.

e. Return result of validation to the forecaster. If process fails for encoding or trend validation, provide the analyst with meaningful error (encoding) or warning (trend) information. If validation passed, publish the TAF.

3. Terminal Forecast Publication Management: The TAF publication function distributes TAF, observation, and alarm data to meet system publication requirements. Major functions:

   a. Accept *validated* TAF and observation data.

   b. Publish TAF and observation data to other regional processing centers.

   c. Send alarm to analyst if forecasted weather element becomes "out of category" with actual weather.

4. Terminal Forecast Collection Management: The TAF Data Management function manages all TFMS data. Major Functions:

   a. Accept TAF, observation, station, and statistical data.

   b. Manage TAF, observation, station, and statistical collections.

   c. Determine TAF weather element category as defined by MAJCOM category and threshold definitions [68].

   d. Compute and update TAF accuracy statistics.

5. Terminal Forecast System Administration: The TAF system administration function permits configuration of the TFMS. Major Functions:

   a. Create and modify weather station data and category definitions.

   b. Schedule TAF accuracy reports generation.

c. Configure TAF collection definitions for each processing center.

3.4.1.3 TAF Validation Issues

The Automated Weather Distribution System (AWDS), which currently produces weather reports at Weather Flights, also has no QC capability. This means that forecasts with typographical errors are submitted into the global network. The TFMS must integrate QC and provide validation facilities for TAF submissions, alerting the forecaster about TAF errors *before* entry into the global network. The TFMS must also provide an alarm facility to notify forecasters that a current forecast has become "out of category" due to recent weather observations.

3.4.1.4 TAF Publication Issues

Since terminal forecasts are used by a wide variety of aviation customers, terminal forecast availability and dissemination is a very important design issue. To ensure the highest availability of *current* terminal forecast information, a replication methodology must be addressed so the system avoids the *centralized bottleneck* issue, which greatly affects system performance and scalability [12]. TAF replication can be modeled using a variety of distributed algorithms. Since terminal forecasts are required in near real-time, performance of this algorithm becomes a very important design consideration.

3.4.1.5 Measuring TAF Data Quality and Accuracy

The TFMS provides *automated* metric collection facilities to report data quality and forecasting accuracy based on submitted TAF/observation data. The first metric collection facility is concerned with collecting statistics based on *static* data components, i.e. data *format*/attribute values, to correct and determine data quality before data dissemination. These measurements serve as an assessment of the overall data quality of submitted TAF/observation products and may also serve as a basis for data warehouse certification [71]. Figure 14 shows the concept behind this approach.

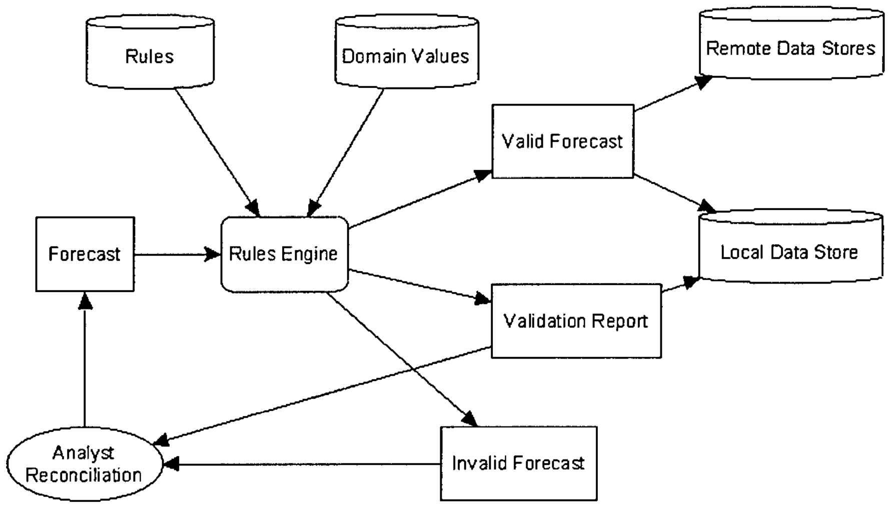


Figure 14: TFMS Static Data Quality

When a forecast is submitted for validation, a rules engine is used to process each forecast attribute, e.g. wind speed. The rules engine uses a set of rules and a set of allowable attribute values to determine the data quality for each attribute. A forecast attribute value passes *format* validation if the attribute's value is a member of the set of domain values defined for that particular attribute, e.g.

forecast type = {amended, corrected, null}. Once all attributes are processed, the forecast is either valid or invalid. Valid forecasts are published to the local OWS data store and other regional sites. Invalid forecasts require reconciliation by the weather analyst to determine why validation failed. In both cases, a validation report collects the result of each rule in the engine and sends the report to the analyst and the local OWS datastore. The weather analyst uses the report to confirm a valid forecast or correct an invalid forecast. The local data store uses the report for statistical purposes to determine data quality for each forecast attribute.

The second metric collection facility is concerned with collecting statistics based on *dynamic* data components, i.e. actual weather conditions versus an analyst's forecasted conditions, to provide the analyst or organization an indication of forecast accuracy. Figure 15 shows the complete TFMS validation and measurement model proposed in this research. When a forecast *or* observation is submitted for validation, a rules engine is used to process each forecast attribute, e.g. wind speed. The rules engine uses a set of rules, domain values, trends, and MAJCOM category definitions to determine the data quality and classification of each weather attribute.

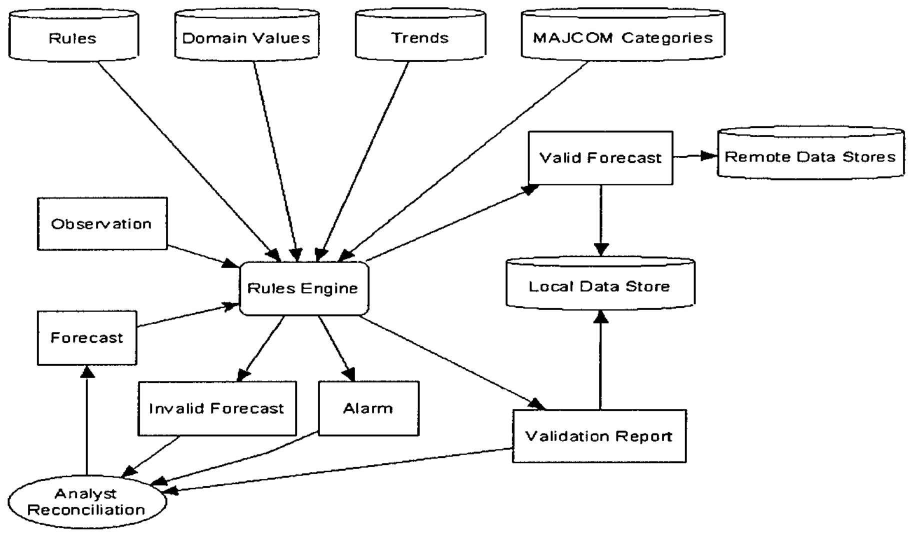


Figure 15: TFMS Validation and Measurement Model

A *forecast* attribute is checked for format, then categorized, e.g. wind speed category A, and finally compared against the most recent observed category (trend) for that specific location. An *observation* attribute is checked for format, then categorized, and its trend information updated accordingly. Once processed, the forecast or observation is either valid or invalid. Valid TAF/observation data is published to the local OWS data store and other regional sites. Invalid forecasts require reconciliation by the weather analyst to determine why validation failed. Reconciliation is also required when an actual weather category differs from the current forecast category for a given weather element, i.e. TAF amend alarm is issued. In all cases, a validation report collects the result of each rule fired in the engine and sends the report to the analyst and the local OWS datastore. The weather analyst uses the report to confirm valid TAF/observation submissions or correct an invalid submission. The local data store uses the report for statistical purposes to determine data quality and record forecast accuracy.

Examples of forecast accuracy metrics include the *percent correct* and *yes/no capability* metrics [68]. As shown in Table 1, categorical skill scores and forecast validation statistics are computed using a two-by-two matrix.

Table 1: Two-By-Two Forecast Validation Matrix

	Forecast "Yes"	**Forecast "No"**
Observed "Yes"	A	B
Observed "No"	C	D

The percent correct and yes/no capability formulas described below are computed using matrix variables A, B, C, and D. These variables are defined for a specific forecast weather Element, e.g. wind speed, as shown in Table 2 [68].

The percent correct metric is the number of forecast hits divided by the total of forecast hits and misses, given by the formula:

$$\textit{Percent Correct} = (A + D) \div (A + B + C + D)$$

The capability yes/no metric is the number of forecast hits (forecast = yes or no) divided by the total of forecast hits and misses (forecast = yes or no) for the weather event, given by the formulae:

$$\textit{Capability (yes)} = A \div (A + B)$$

$$\textit{Capability (no)} = D \div (C + D)$$

Table 2: Matrix Element Definitions

Matrix Element	Definition
A	Number of times that a categorized weather element was forecast to occur (forecast = Yes) and then actually observed (observed = Yes) (regarded as a forecast "hit")
B	Number of times that a categorized weather element was not forecast to occur (forecast = No) but then actually observed (observed = Yes) (regarded as a forecast "miss")
C	Number of times that a categorized weather element was forecast to occur (forecast = Yes) but was not actually observed (observed = No) (regarded as a forecast "miss")
D	Number of times that a categorized weather element was not forecast to occur (forecast = No) and was not observed (observed = No) (regarded as a forecast "hit")

3.4.2 Information System Model

An information system model describes system objects and shows how these objects are used or processed by the software system [6 and 27]. There are three analysis tools that are very useful in decomposing a complex distributed object system. The first is the software system schema, a.k.a. an object/entity-relationship model, which serves to identify software system objects, showing how they collaborate within the information system. The second is the information flow model, which shows how data is transformed through the software system [6]. The final tool is the transaction model, which decomposes system transactions or use case scenarios, into object-specific behavior [27].

3.4.2.1 TFMS Schema

In constructing the software system schema, data and control objects are identified and their relationships analyzed [6]. The following criteria are useful for categorizing objects in the software system schema [76]:

1. I/O Objects: These objects correspond to abstractions dealing with system I/O functions, e.g. network I/O, input sensors, and output controllers.
2. User Role Objects: Represents a type of user or system action interfacing with the system, e.g. submitting a forecast for validation/publication.
3. Control Objects: An *active* object that has *state* and controls the behavior of other objects or functions, e.g. publication and validation agents.
4. Data Abstraction Objects: Data stores, e.g. weather station and validated forecast persistent data.
5. Algorithm Objects: Encapsulates an algorithm in the problem domain, e.g. primary copy, pipeline and CORBA Event Service publication algorithms.

As shown in Figure 16, the TFMS is composed of an organization, agents, alarms, reports, and system settings. The AF Weather organization contains regions, major commands, and weather encoding instructions applicable to all weather stations [59]. Regions consist of weather stations while major commands contain weather stations and define weather validation categories specific to the command's requirements. Each validation category, e.g. wind speed, contains a number of thresholds that further subdivide that category, e.g. category A, B, C... etc.

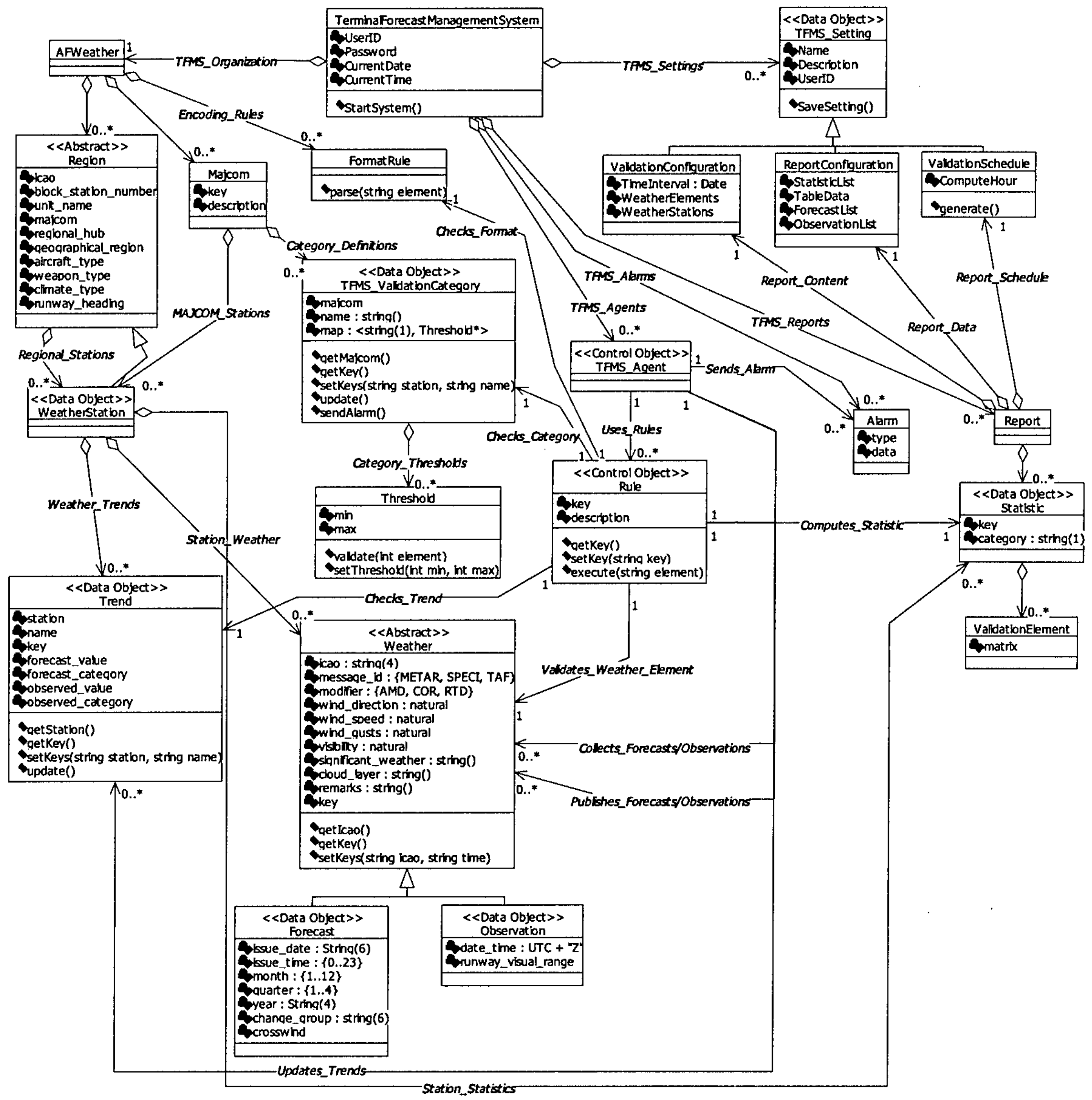


Figure 16: TFMS Object-Relationship Diagram

Each weather station contains unique weather, trends, and statistics. TFMS agents perform a number of operations within the system (§3.4.2.3 Role of TFMS

Agents). For instance, a validation agent checks a weather element's format and validates the element against station trends. TFMS alarms notify the weather analyst of a system condition requiring their attention. For instance, an amend alarm for a given location would indicate the latest forecast recorded for that station is out of category with the latest observation on record.

TFMS reports consist of the analyst's desired time interval, weather stations, and weather elements involved in the report. TFMS system settings are configured so each analyst may customize the environment to individual preferences - accomplished by using the network logon user id. While Figure 16 shows what objects, relationships, and message paths are used with the TFMS schema, it really doesn't show how information is processed. In the next section, data flow diagrams are used to show how information flows through the system.

3.4.2.2 Information Flow

Data Flow Diagrams (DFD) help reduce complexities for software engineers by providing a system view as a network of processes connected by data flows [27]. There are many uses for a DFD besides simply showing information flow within a software system. A DFD with formal semantics can model critical software functions and determine the serializability of distributed transactions (§2.5.3 Serializability) to ensure design correctness [39]. In this section, a formal semantics approach is not taken. A DFD is simply used here to decompose the information model to reduce the inherent complexities with distributed object system design.

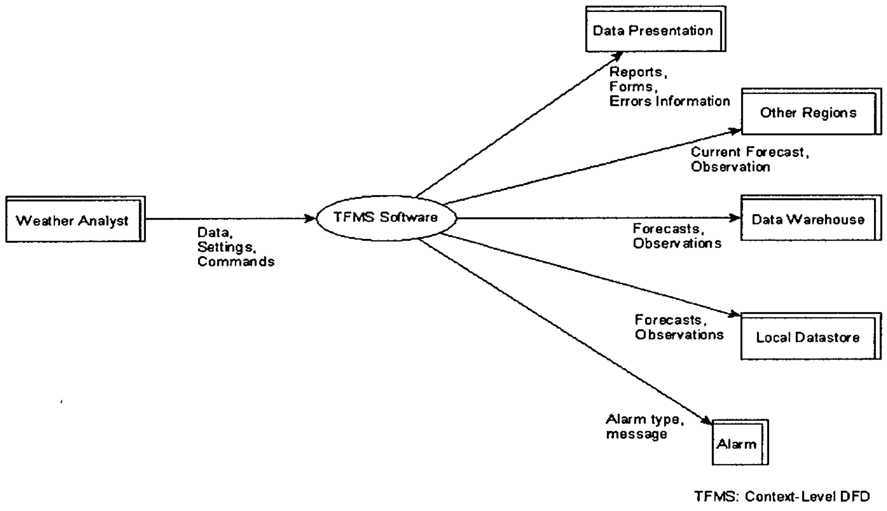


Figure 17: TFMS Context-Level Data Flow Diagram

Figure 17 shows the starting or context -level for function decomposition. The main focus of the context-level is to show the producers and consumers of information flow [6]. The TFMS identifies the weather analyst as the producer and the user interface, other OWS sites, AFW data warehouse, local datastore, and alarms as the consumers of information.

In Figure 18, the TFMS context-level DFD is decomposed into a level one DFD to further define the TFMS software system. As Figure 18 shows, there are four primary processes: 1) Configure stations, settings, and categories; 2) Process forecasts; 3) Process observations; and 4) Generate Reports. Forecast, observation, statistic, setting, station, category, rule, format, and trend data objects are also shown combined into one datastore; this datastore is further decomposed along with the rest of the system in subsequent diagrams.

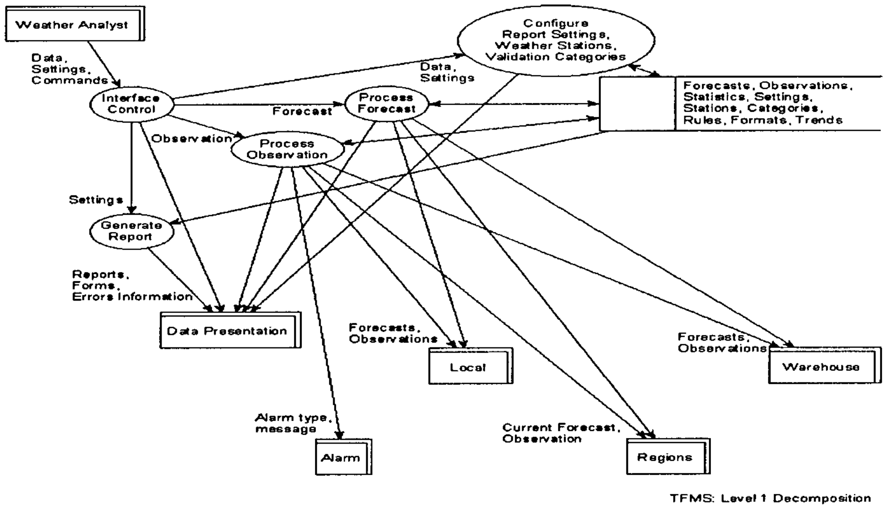


Figure 18: TFMS Level One Data Flow Diagram

The *process forecast* and *process observation* processes identified in the level one DFD are further decomposed in Figures 19 and 20.

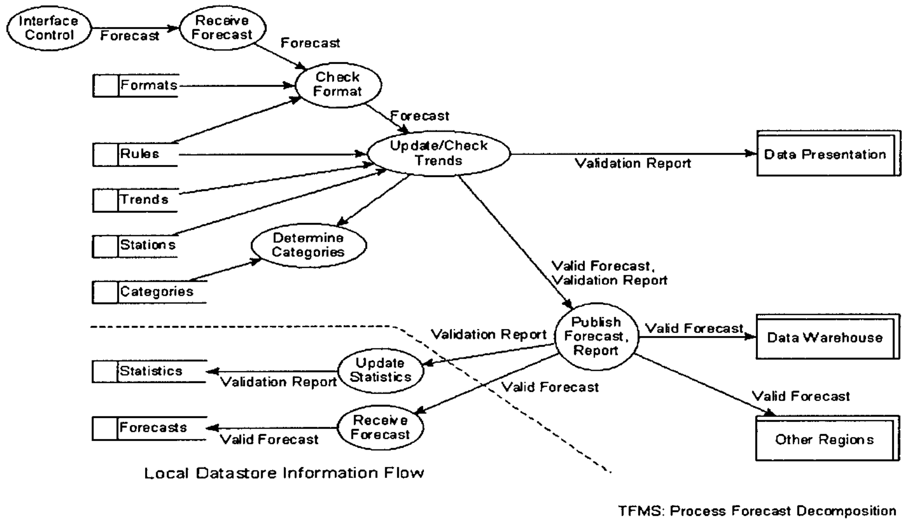


Figure 19: TFMS Process Forecast Data Flow Diagram

In Figure 19, a forecast is received from the user interface. The forecast's format and trend is then validated. If the forecast fails validation, the forecast and a validation report are returned to the weather analyst. The validation report is also published to the local data store to update *format* statistics. If the forecast passes validation, the forecast is stored locally then published to other OWS sites. The validation report is also published to the local data store to update *format* and *accuracy* statistics.

In Figure 20, observations are processed in much the same way. The primary difference is the check/update trend process. An alarm is generated if the observed condition is out of category with the current forecast condition.

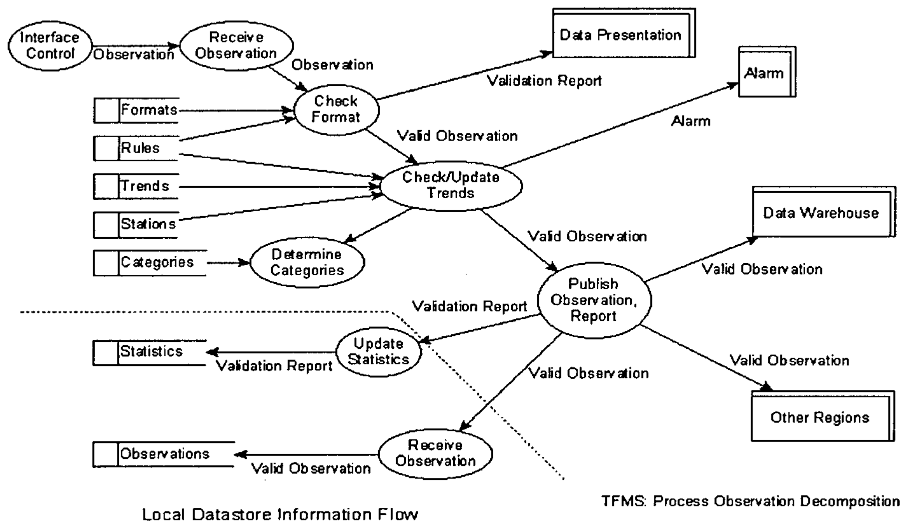


Figure 20: TFMS Process Observation Data Flow Diagram

3.4.2.3 Role of TFMS Agents

In the TFMS schema, a distinction was not made on the roles a TFMS agent can take. This section provides a brief discussion of major TFMS control agents and their responsibilities.

Role of Validation Agents: Validation agents ensure all TAF data values are checked for proper encoding IAW AFM 15-124, Meteorological Codes [57]. In addition, validation agents also check a forecast's weather element value against the most recently recorded observation value for that particular element. Validation agents will use a number of different data validation methods to accomplish this task. Since each terminal site possesses unique weather patterns, a validation agent must ensure it uses the appropriate trend data for each site. Because validation agents require observation trend data as part of the validation process, this data must be locally accessible. Validation agents also document validation results in the form of a validation report for each forecast or observation attribute. Validation agents coordinate with publication agents to disseminate weather, reports, and alarms.

Role of Publication Agents: Since validation is performed *prior* to TAF entry into the distributed system (§3.3 Proposed Forecast Generation), the data is *assumed* correct when it reaches a publication agent in the same address space. Publication agents distribute weather, report, and alarm data to meet system availability, functionality, and performance requirements. The specific replication facility for weather data distribution is subject to experimentation based on the *read-only* characteristic of a validated TAF or observation. TAF/observation publications must be written to stable storage prior to replication [9 and 12]. Publication agents coordinate with collection and metric agents to deliver system products.

Role of Collection Agents: Collection agents build and maintain data object cache collections (stations, forecasts, observations, and statistics) and usually consist of two other system agents - query and notification agents [41 and 43]. The query agent is responsible for building the cache collection based on SQL or object query language (OQL) data manipulated language (DML) declarations over the underlying datastore [9 and 44]. The notification agent is responsible for receiving weather data from other regions, so it's used as a *sink* for CORBA weather events. In the local datastore, collection agents receive *validated* weather data from a publication agent and ensure cache collections are updated accordingly.

Role of Metric Agents: Metric agents are responsible for receiving validation reports from publication agents, and updating regional statistics using report data. As mentioned above (§3.4.1.5 Measuring TAF Data Quality and Accuracy), validation reports may also be used to certify the data quality of weather data before entry into the data warehouse [71]. For example, if AFWA requires a 99 percent data quality percentage for the wind speed attribute, and validation reports show a 92 percent format validation rate, then AFWA or the associated OWS must analyze invalid forecast reports to find the root cause for the wind speed validation failures.

3.4.2.4 Transaction Model

The transaction model describes task decomposition and *interactions* among objects identified in the TFMS schema and DFD models [13]. This is analogous to system interaction diagrams. Based on these diagrams and descriptions, we can probe for distributed processing opportunities in the system to maximize computing resource efficiency [15]. This model also provides the basis for two other issues concerning system design -- system test plan and user interface development. The system test plan is designed to aid the validation process for integration testing while user interface design is primarily derived from the transaction model decomposition [27].

The primary reason for choosing this design method over use case scenario methods is it captures the same information in a more understandable format, while also addressing integration testing and user interface design issues as well, i.e. it ties all high-level aspects of the system together very nicely.

3.4.2.4.1 Task Decomposition

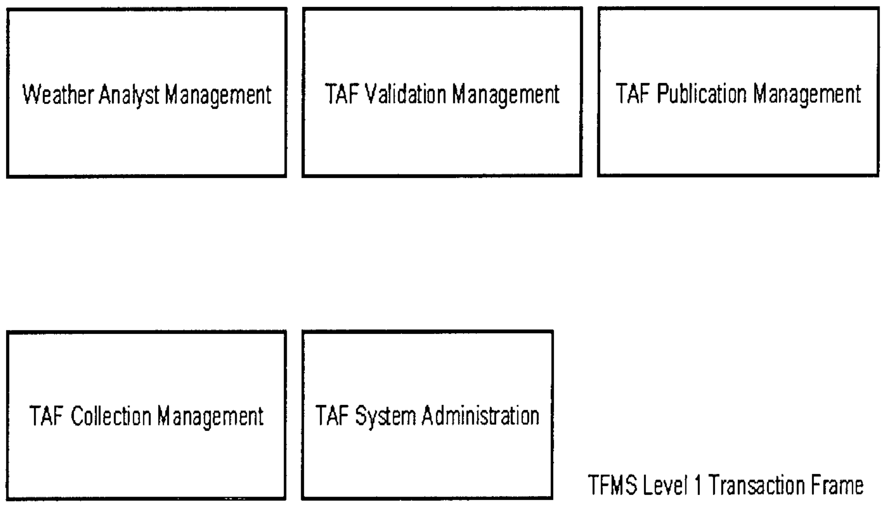

Weather Analyst Management
TAF Validation Management
TAF Publication Management
TAF Collection Management
TAF System Administration
TFMS Level 1 Transaction Frame

Figure 21: TFMS Transaction Model

Figure 21 shows top-level actions for the TFMS distributed system based on TFMS functional requirements (§3.4.1.2 Functional Description). Figure 22 shows further decomposition applied to *weather analyst management*. Each action in Figure 22 is refined until object-level interactions are exposed. In this section, the *"submit forecast"* action is further decomposed to demonstrate the methodology.

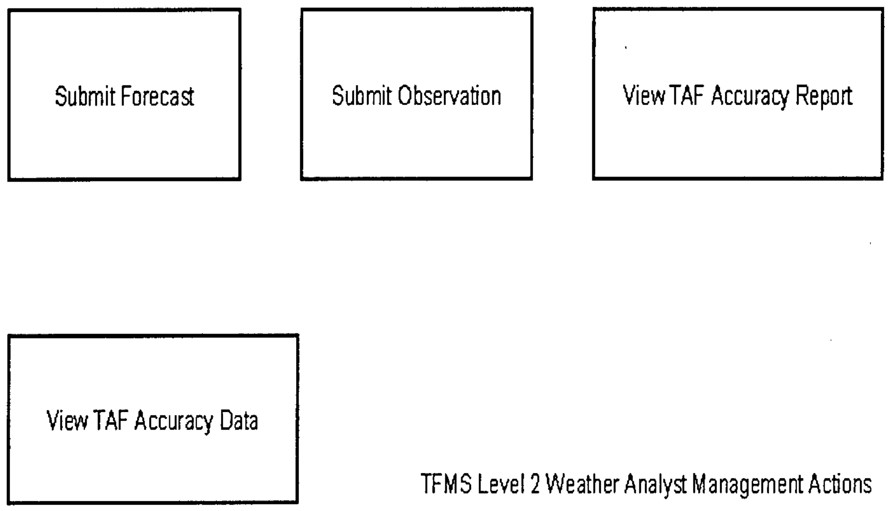


Figure 22: Weather Analyst Management Transaction Model

Figure 23 shows a suitable breakdown for the *"submit forecast"* transaction. The diagram shows a complete sequence of operations needed to carry out the action and the major objects involved with each operation to form the transaction. Figure 23 also serves as one validation test case in the TFMS system test plan [6 and 27].

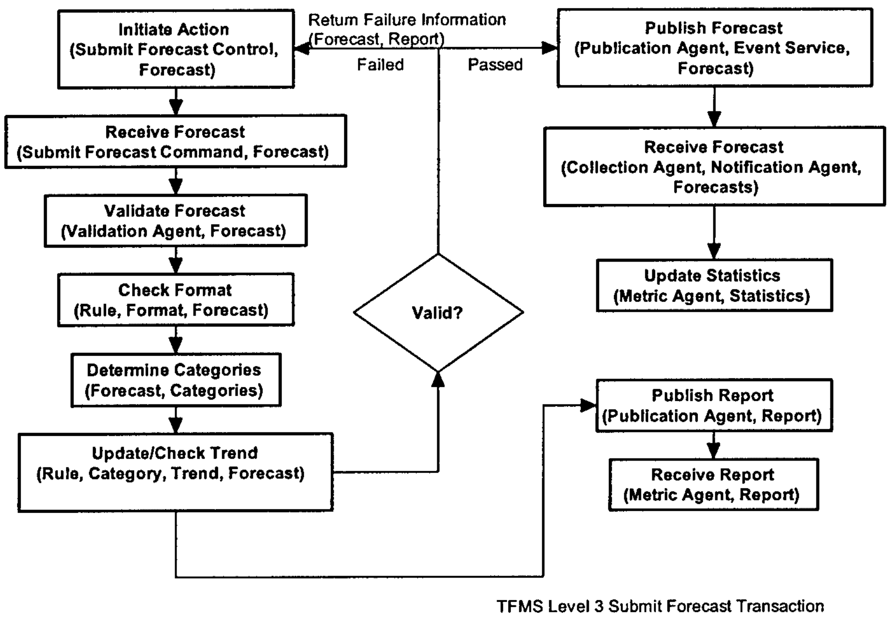

Initiate Action (Submit Forecast Control, Forecast)
Return Failure Information (Forecast, Report)
Failed
Passed
Publish Forecast (Publication Agent, Event Service, Forecast)
Receive Forecast (Submit Forecast Command, Forecast)
Receive Forecast (Collection Agent, Notification Agent, Forecasts)
Validate Forecast (Validation Agent, Forecast)
Update Statistics (Metric Agent, Statistics)
Check Format (Rule, Format, Forecast)
Valid?
Determine Categories (Forecast, Categories)
Publish Report (Publication Agent, Report)
Update/Check Trend (Rule, Category, Trend, Forecast)
Receive Report (Metric Agent, Report)
TFMS Level 3 Submit Forecast Transaction

Figure 23: TFMS Submit Forecast Transaction

This is a very simple, but powerful tool when performing distributed data analysis [27]. From this simple flowchart, processes uncovered during DFD modeling (§3.4.2.2 Information Flow) are mapped to objects identified during TFMS schema definition (§3.4.2.1 TFMS Schema).

3.4.2.4.2 Object-Level Behavior

The object-behavior model further decomposes the task decomposition model to arrive at object-level method descriptions that perform application logic [27]. Figure 24 shows the "submit forecast" transaction further decomposed to the object level for the publication agent. All objects are analyzed in the same fashion.

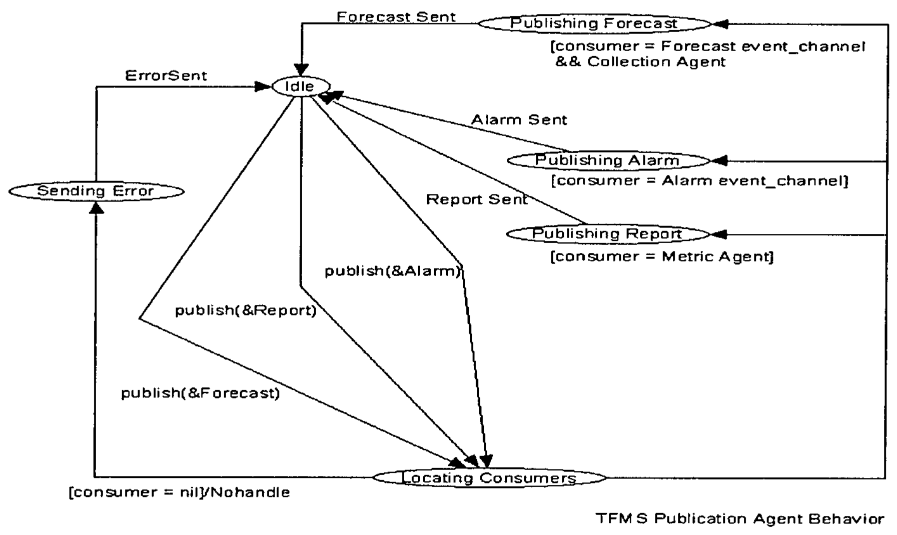


Figure 24: Publication Agent Behavior

Based on this decomposition, the publication agent reacts to three external events: overloaded alarm, report, and forecast method invocations. The publication agent first locates consumers for the particular event. If no consumers are on-line (consumer = nil), an error is returned to the caller. If consumers are on-line, the publication agent changes state to publish the forecast, report, or alarm. One distributed insight into this state diagram is the guard condition for publishing forecasts (consumer = forecast event channel and collection agent). If the publication operation is communication-intensive (bulk forecast transfers), a further decomposition could divide the operation into two private methods: a local communication operation (collection agent) and a remote communication operation (forecast event channel). Each operation would be multi-threaded so the communication load is divided to increase server-side performance and improve operation execution time [21 and 47].

3.4.2.5 Granularity

Determining granularity in a distributed system has a direct impact on system interface design, performance, and scalability [10, 15, and 21]. The TFMS is centered on validation and distribution of *terminal forecasts and observations;* therefore, a fine level of granularity is appropriate for these objects, i.e. object-level granularity. Currently, passing an object by value is supported in CORBA version 2.3, but not widely implemented [21]. Since interoperability is always a major consideration, there are two choices: a structure may be defined in CORBA IDL to meet necessary functionality for data handling purposes or a reference to an interface may be used to retrieve data members.

It's very important to define rules for grouping objects and/or components into subsystems. In this paper, subsystem, module, and package are used interchangeably. Cohesion and coupling criteria are used to partition the system into subsystems, components, or objects capable of autonomous operation [6].

Cohesion Criteria: Objects may be functionally, sequentially, or temporally grouped into subsystems or components. Functional cohesion is the strongest criteria for module structuring since it means the objects perform similar functions in the system, e.g. validation or publication functions. Temporal cohesion is the weakest module structuring criteria because this indicates that *different* objects may execute at the same time (concurrently) and *should* be considered as separate, autonomous system entities. In some cases, the functional aspects of the objects may warrant their allocation to the same module.

Coupling Criteria: Coupling refers to dependencies in the distributed system. There are many types of coupling [6], but in a distributed system, *loose* coupling is desired to ensure the autonomy of physical processing nodes, i.e. no subsystem, component, or object depends upon another system entity to carry out its function in the system. For example, the forecast publication module publishes forecasts. If the publication module were designed to have knowledge or *dependencies* of a *particular* forecast validation module, then the system would quickly lose its fault tolerance capabilities of simply loading another validation module to continue operation and improve overall system availability [10]. This simple scenario also speaks volumes about system *flexibility* and the dynamic capacity of the software system in general [8].

The TFMS validation, metric, and publication functions are other candidates for distribution, since rules and statistical operations may be computationally expensive while the publishing operation may be an expensive communication operation. For instance, to *validate* a forecast sequentially, the validation agent checks each weather element using a business rule for each element. If these business rules should become *computationally* expensive, then distributed processing using all available platforms needs to be considered as a solution [15]. Another consideration is bulk validations. If the validation execution time is acceptable for a single forecast, but excessive in the case of ten forecasts, then distributed processing using a coarser granularity (forecast granularity as opposed to weather element granularity using individual rules) may be more appropriate. For finer granularity, validation agents would multithread expensive business rules and obtain an interface reference on a separate CORBA server to accomplish the same rule on a different processor. Another approach would be to implement *each* weather attribute as a *dependent object* [72]. A dependent object is effectively a data *wrapper* that encapsulates each weather element and applies specific behavior and rules to that weather element.

For coarser granularity, the validation agent would send forecasts to another validation function for processing on a separate CORBA server. In both cases, the validation function must be measured and the appropriate processing granularity determined. Partitioning the TFMS validation function in this manner provides the flexibility to distribute the validation process over any number of local OWS processor resources (§3.5.2.5 Validation Module).

3.5 Distributed Object System Design

The primary purpose of distributed object system design is to transform the software requirements specification into an architectural, interface, data, and procedural design that is traceable to customer's requirements. Other objectives pertain to software quality and extensibility [5, 6, and 29].

Distributed object system design also considers a *scalability model,* which separates design and implementation concerns based upon the scale of the problem we're trying to solve, e.g. publishing forecasts to all OWS regional processing centers for maximum forecast availability is an enterprise-level problem. The scalability model helps us decide if we should use a system-provided service, e.g. OMG Event Service, or apply a known framework or design pattern to solve the problem [14]:

1. Global: Interoperability between organizations, e.g. open systems.
2. Enterprise: Interoperability *within* an organization, e.g. organizational hardware and software standards.
3. System: Interoperability between applications.
4. Subsystem: Developing applications that meet a specific user requirement.
5. Framework: Developing generic solutions of cooperating *components* that can be instantiated to solve a specific subsystem design issue.
6. Component: Developing generic solutions of cooperating *objects* to solve recurring problems.
7. Object: Concerned with *code reuse* more than design reuse.

3.5.1 System Partitioning

System partitioning defines an architectural structure and the relationships among major elements in the software system [6]. As shown in Figure 25, the validation, publication, and data functions are placed in separate modules that provide clean separation of functionality coupled with a *coarse* level of granularity (§3.4.2.5 Granularity), which reduces communications costs [1 and 15]. TFMS system modules (subsystems) are then partitioned along client, business, and data boundaries as shown in Figure 25. This partitioning follows an N-tier modeling methodology [2, 4, 7, and 8]. In the user services layer, weather analyst functions are grouped into a client module. The business layer contains forecast validation logic, grouped into a validation module; publication functions, grouped into a publication module; and data classification logic, grouped into a classification module.

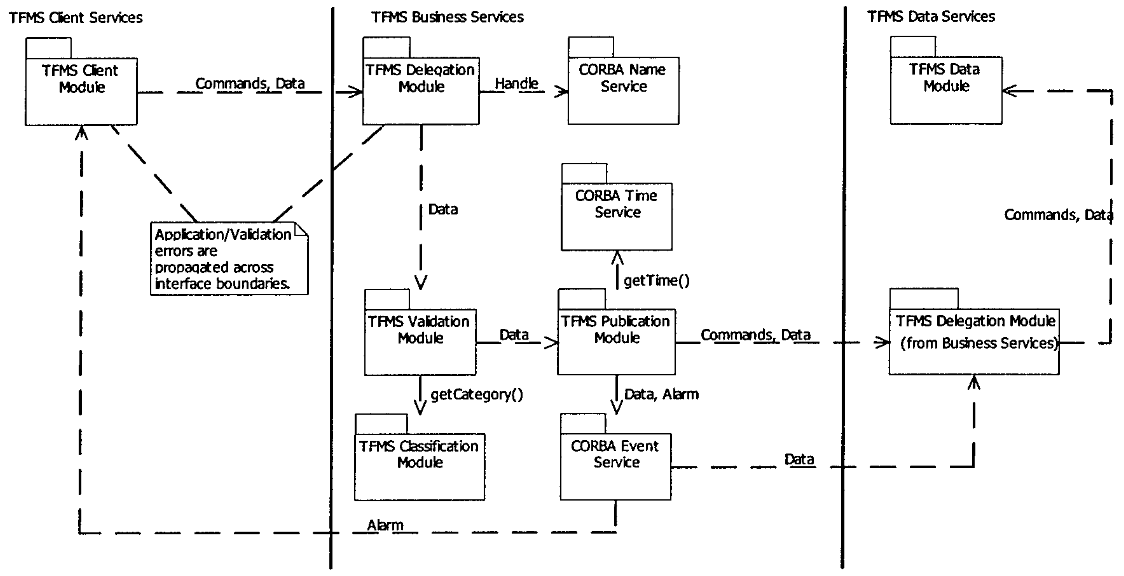


Figure 25: TFMS Modular View

It's during the system-partitioning phase that global, enterprise, and system design patterns are introduced. The system-level architecture is the enduring structure that survives component modification, evolution, and interoperability over the system's lifecycle [14]. Using the CORBA Name, Time, and Event services to implement application requirements provides reuse at the architectural level while controlling system evolution and complexity. The data layer contains collection management functions and statistical computation grouped into a data module. System administration functions are integrated according to the affected module. Internal design for each module is discussed below. The delegation module is reused in the business and data layers (§3.5.2.4 Delegation Module).

3.5.2 System Module Design

Modular design addresses the issues of interface, data, and procedural design. Task architecture and concurrency issues are also addressed during system module design. If the module is providing services to many clients, concurrent/synchronization issues become a large part of the design effort to improve throughput [76]. During modular design, design patterns and frameworks are also considered and applied where appropriate. A key OO principle is very noticeable in distributed object system development: program to an interface, not to an implementation [13 and 14]. Adhering to this principle ensures clients remain *unaware* of *specific object types* and underlying *object implementations*. This reduces dependencies within the system aiding reuse [13].

3.5.2.1 Thread Architecture

Thread structuring criteria are used to map threads to individual objects or groups of objects in the subsystem architecture. Some rules to consider when specifying the thread architecture [76]:

1. I/O Objects: Objects interacting with I/O devices, e.g. network or disk, are mapped to a distinct thread.
2. Internal Objects: Periodic, asynchronous and user role objects are mapped to a distinct thread. Periodic objects that execute in the same time period may be grouped into the same thread if they're not different priorities, i.e. by temporal cohesion criteria.
3. Cohesion Criteria: Objects meeting control, sequential or temporal cohesion are mapped or grouped into distinct threads. These usually include control/command objects, computationally expensive *sequential* operations, and periodic operations as mentioned previously.

3.5.2.2 Data Architecture

Data structuring criteria are used to map data stores in the subsystem architecture. A few data abstractions to consider [76]:

1. I/O Abstraction Interfaces: Provide I/O interface definitions to encapsulate device/access details.
2. Data Abstraction Interfaces: Encapsulate sequential, e.g. vectors and lists, and associative, e.g. maps and sets, container implementations.

3. Algorithm Abstraction Interfaces: Encapsulate all application algorithms in objects to hide algorithm details.

Finally, once the task and data architectures are defined and integrated to develop the subsystem or component architecture, the interface is defined. The subsystem or component's interface characterizes the complete set of requests that can be sent to the module [13]. The rest of this section discusses TFMS design, module by module using the software requirement specification as input. An important consideration to any modeling methodology is its comprehensibility. Too much detail clutters the model, making it incomprehensible [27].

3.5.2.3 Client Module

Figure 26 shows the client module design. The weather analyst communicates with the TFMS distributed system using the CORBA interfaces defined in the TFMS Delegation module. If an analyst wants to view statistics, schedule statistical computation, or submit a set of forecasts/observations for validation and publication, these interfaces characterize a complete set of operations that can be invoked on an object, component, or module [6].

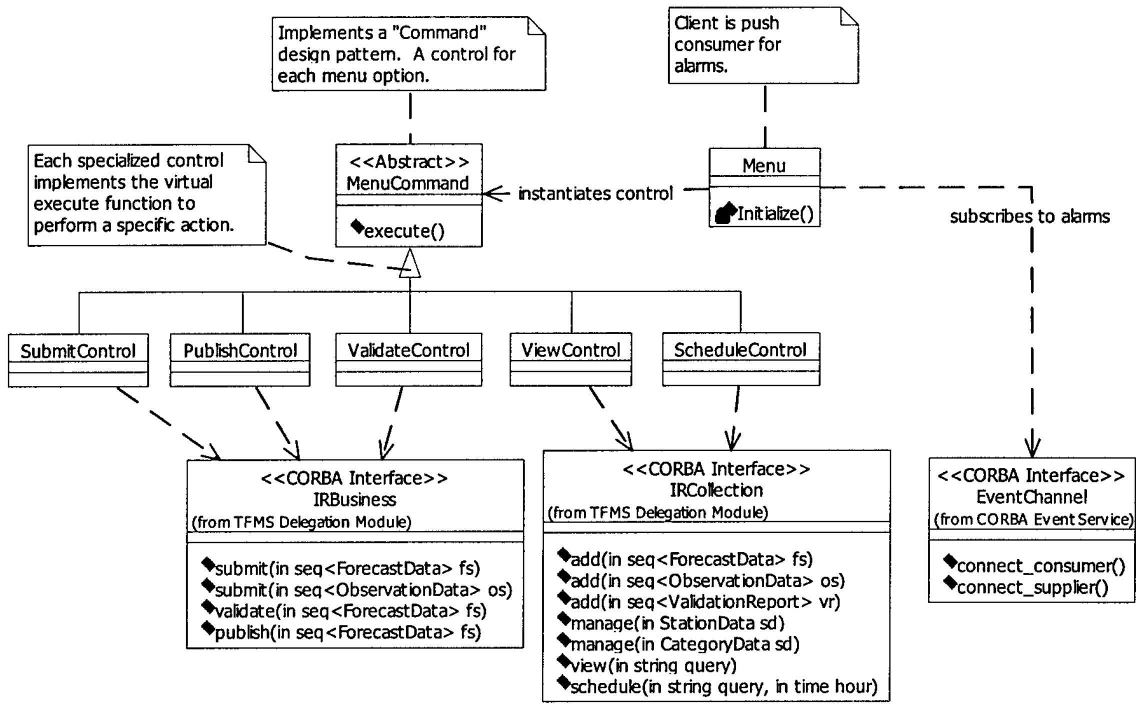
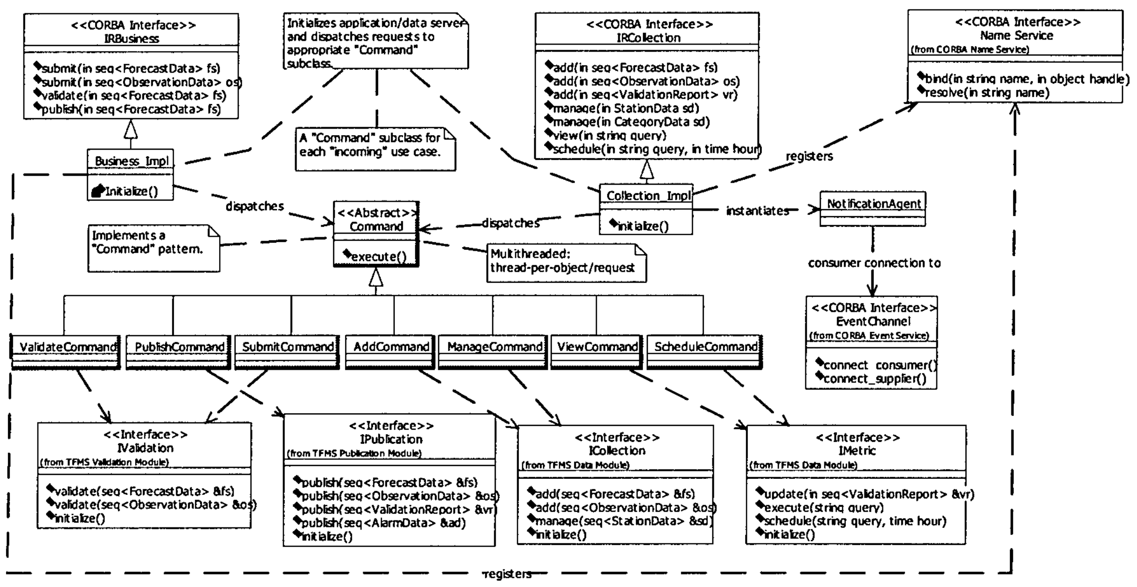

Implements a "Command" design pattern. A control for each menu option.
Client is push consumer for alarms.
Each specialized control implements the virtual execute function to perform a specific action.
<<Abstract>> MenuCommand
execute()
instantiates control
Menu
Initialize()
subscribes to alarms
SubmitControl
PublishControl
ValidateControl
ViewControl
ScheduleControl
<<CORBA Interface>> IRBusiness (from TFMS Delegation Module)
submit(in seq<ForecastData> fs)
submit(in seq<ObservationData> os)
validate(in seq<ForecastData> fs)
publish(in seq<ForecastData> fs)
<<CORBA Interface>> IRCollection (from TFMS Delegation Module)
add(in seq<ForecastData> fs)
add(in seq<ObservationData> os)
add(in seq<ValidationReport> vr)
manage(in StationData sd)
manage(in CategoryData sd)
view(in string query)
schedule(in string query, in time hour)
<<CORBA Interface>> EventChannel (from CORBA Event Service)
connect_consumer()
connect_supplier()

Figure 26: TFMS Client Module

As shown in Figure 26, the client module is very simple and its functionality is easily traceable back to user requirements (§3.4.1.2 Functional Description). When the client module initializes, the program connects to the CORBA Event Channel to subscribe to alarm data. The analyst is then presented with a menu of options similar to the ones shown in the transaction model. When the observer chooses an option, a *specialized* menu control is *instantiated* to execute user interface logic pertaining to that *system action.* This is an application of the well-known *command* design pattern that removes tight coupling between objects to reduce object dependencies [6 and 13]. We can easily extend our user interface by defining new control subclasses to add new window or menu functionality.

3.5.2.4 Delegation Module

The TFMS delegation module is shown in Figure 27. The purpose of the delegation module is to separate CORBA and platform-specific thread code from business logic. This approach improves physical design, isolates portability problems and CORBA errors to this module, reduces compile and link times by limiting CORBA source file dependencies, and limits the need for developers to be proficient in using a specific CORBA language mapping [21 and 22].



<<CORBA Interface>>
IRBusiness
submit(in seq<ForecastData> fs)
submit(in seq<ObservationData> os)
validate(in seq<ForecastData> fs)
publish(in seq<ForecastData> fs)
Initializes application/data server and dispatches requests to appropriate "Command" subclass.
<<CORBA Interface>>
IRCollection
add(in seq<ForecastData> fs)
add(in seq<ObservationData> os)
add(in seq<ValidationReport> vr)
manage(in StationData sd)
manage(in CategoryData sd)
view(in string query)
schedule(in string query, in time hour)
<<CORBA Interface>>
Name Service
(from CORBA Name Service)
bind(in string name, in object handle)
resolve(in string name)
A "Command" subclass for each "incoming" use case.
Business_Impl
initialize()
registers
Collection_Impl
initialize()
dispatches
<<Abstract>>
Command
execute()
dispatches
instantiates
NotificationAgent
Implements a "Command" pattern.
Multithreaded: thread-per-object/request
consumer connection to
<<CORBA Interface>>
EventChannel
(from CORBA Event Service)
connect_consumer()
connect_supplier()
ValidateCommand
PublishCommand
SubmitCommand
AddCommand
ManageCommand
ViewCommand
ScheduleCommand
<<Interface>>
IValidation
(from TFMS Validation Module)
validate(seq<ForecastData> &fs)
validate(seq<ObservationData> &os)
initialize()
<<Interface>>
IPublication
(from TFMS Publication Module)
publish(seq<ForecastData> &fs)
publish(seq<ObservationData> &os)
publish(seq<ValidationReport> &vr)
publish(seq<AlarmData> &ad)
initialize()
<<Interface>>
ICollection
(from TFMS Data Module)
add(seq<ForecastData> &fs)
add(seq<ObservationData> &os)
manage(seq<StationData> &sd)
initialize()
<<Interface>>
IMetric
(from TFMS Data Module)
update(in seq<ValidationReport> &vr)
execute(string query)
schedule(string query, time hour)
initialize()
registers

Figure 27: TFMS Delegation Module

The module is initialized in either a business or data server mode, further promoting reuse within the system. Two CORBA IDL interfaces: *IRBusiness* and *IRCollection* are defined to permit this flexible arrangement. A *fine* level of granularity is implemented by defining CORBA data structures for weather and report data. Defining CORBA sequences for data structures as shown for the "submit" and "add" overloaded operations supports bulk operation requirements.

To implement CORBA IDL-declared operations, we define a *concrete* subclass that implements the C++ virtual method definitions generated in a header file by the IDL compiler. The *Business_Impl* and *Collection_Impl* classes perform this function and *dispatch* a command subclass for each CORBA operation invoked. Command subclasses also implement the command design pattern (a.k.a. coordinator and transaction patterns) and serve as client-processing agents; providing uniform, consolidated access to a number of TFMS services [13 and 14]. To improve server throughput, command subclasses are multithreaded (§4.2.2.3 Server-Side Performance) [21]. The notification agent is also part of this module, instantiated by the *Collection_Impl* class as a *sink* for TAF/observation publication events using CORBA's Event Service. As shown in Figure 27, the delegation module only communicates with other TFMS module interfaces, so this module is completely decoupled from other TFMS modules [6, 13, and 14].

3.5.2.5 Validation Module

The TFMS validation module is shown in Figure 28. The purpose of the validation module is to validate forecasts by checking encoding format and comparing current forecast categories against the latest weather observation category. As Figure 28 shows, rule, format, and trend containers are mapped from the TFMS schema (§3.4.2.1 TFMS Schema). To obtain maximum reuse, efficiency, and reliability, the validation module uses C++ standard template library (STL) *map* and *stack* containers [9 and 70]. The benefits of using generic components, e.g. reusability, reliability, reduced development time, are widely known [51 and 70].

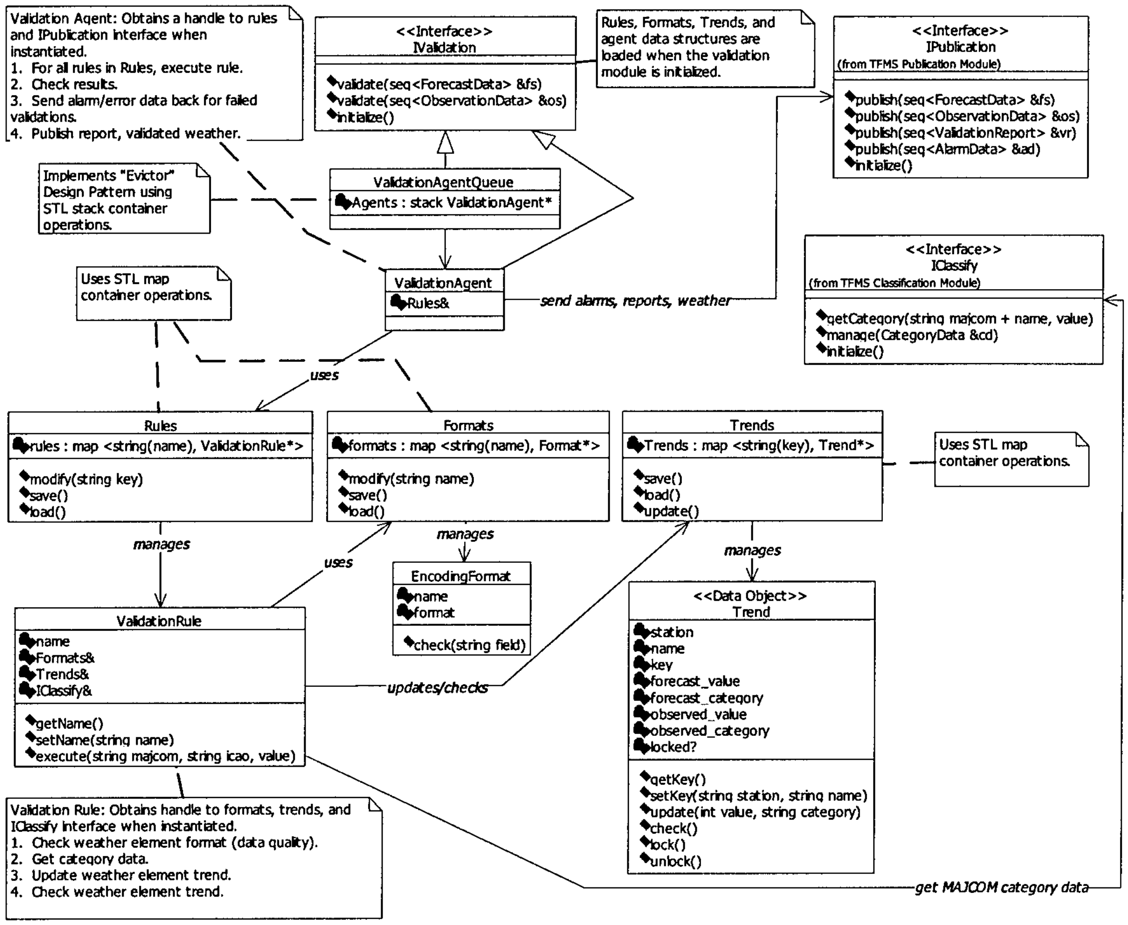

Validation Agent: Obtains a handle to rules and IPublication interface when instantiated.
1. For all rules in Rules, execute rule.
2. Check results.
3. Send alarm/error data back for failed validations.
4. Publish report, validated weather.
<<Interface>> IValidation
validate(seq<ForecastData> &fs)
validate(seq<ObservationData> &os)
initialize()
Rules, Formats, Trends, and agent data structures are loaded when the validation module is initialized.
<<Interface>> IPublication (from TFMS Publication Module)
publish(seq<ForecastData> &fs)
publish(seq<ObservationData> &os)
publish(seq<ValidationReport> &vr)
publish(seq<AlarmData> &ad)
initialize()
Implements "Evictor" Design Pattern using STL stack container operations.
ValidationAgentQueue
Agents : stack ValidationAgent*
ValidationAgent
Rules&
send alarms, reports, weather
Uses STL map container operations.
<<Interface>> IClassify (from TFMS Classification Module)
getCategory(string majcom + name, value)
manage(CategoryData &cd)
initialize()
uses
Rules
rules : map <string(name), ValidationRule*>
modify(string key)
save()
load()
Formats
formats : map <string(name), Format*>
modify(string name)
save()
load()
Trends
Trends : map <string(key), Trend*>
save()
load()
update()
Uses STL map container operations.
manages
uses
manages
manages
EncodingFormat
name
format
check(string field)
ValidationRule
name
Formats&
Trends&
IClassify&
getName()
setName(string name)
execute(string majcom, string icao, value)
updates/checks
<<Data Object>> Trend
station
name
key
forecast_value
forecast_category
observed_value
observed_category
locked?
getKey()
setKey(string station, string name)
update(int value, string category)
check()
lock()
unlock()
Validation Rule: Obtains handle to formats, trends, and IClassify interface when instantiated.
1. Check weather element format (data quality).
2. Get category data.
3. Update weather element trend.
4. Check weather element trend.
get MAJCOM category data

Figure 28: TFMS Validation Module

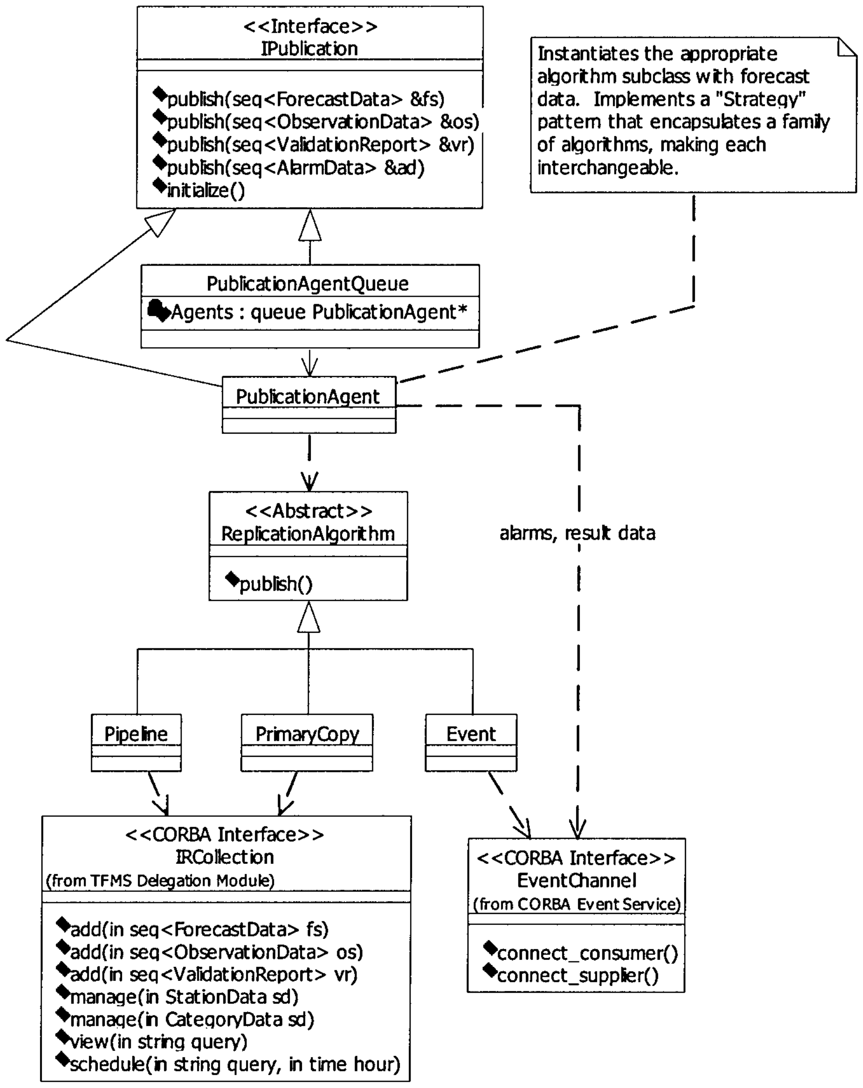
The *ValidationAgentQueue* container holds idle validation agents to reduce execution time. This is a simple implementation of the *evictor* design pattern that reduces the delay involved with creating validation agents for each request. The evictor design pattern also provides memory management benefits, where agents are evicted from the queue if they're idle too long [21]. The validation, publication, and data modules all use the evictor pattern to reduce the cumulative object creation delay throughout the TFMS.

The validation module does have *concurrency* issues with the trend container. When the module receives a set of TAF/observations, all station trends must be updated prior to checking if the current forecast's weather element is "out of category" with the current observation. There are two ways to implement lock granularity in this case: at the *container* level or at the trend *object* level. For container-level locking, the validation agent gets all category update data from the classification module as a bulk request, obtains a *lock* on the trend container, *updates* the trends, and then *unlocks* it. In this case, rules are locked out until the update is performed and the validation agent unlocks the lock variable for the container. *Starvation* is possible in the case of very large bulk validation requests. Implementing the lock at the object level would require only a few minor changes. The *rule* now obtains the category update from the classification module, obtains a lock on an individual trend object, updates and checks the weather element, and then unlocks it. In this case, other rules are locked out only if they try to access a locked trend object. Starvation is very unlikely since the weather station ICAO identifier is used as part of a unique map key, so a given TAF/observation update operation will only affect the trend objects for that weather station and element. This design greatly increases application concurrency by a factor of the size of the trend container, while keeping lock operation complexity the same. Figure 28 shows the trend object with the lock and unlock operation additions to support operation serialization functionality.

3.5.2.6 Publication Module

The TFMS publication module is shown in Figure 29. The purpose of the publication module is to publish validated weather to meet performance, availability, and AFW dissemination requirements IAW AFMAN 15-111 [67]. This module is also responsible for publishing TFMS alarms to the CORBA Event Channel and validation reports to the local OWS data module.

Since the publication module already uses the CORBA Event Service to send TAF/observation data to other OWS sites, alarm functionality was moved from the validation module to improve physical design by removing all CORBA source file dependencies from the validation module (§3.5.2.4 Delegation Module).. The publication module also implements a "Strategy" design pattern that encapsulates a family of algorithms, i.e. primary copy, pipeline, event, making the experimentation process simply a matter of algorithm selection, as opposed to changing source code [13].

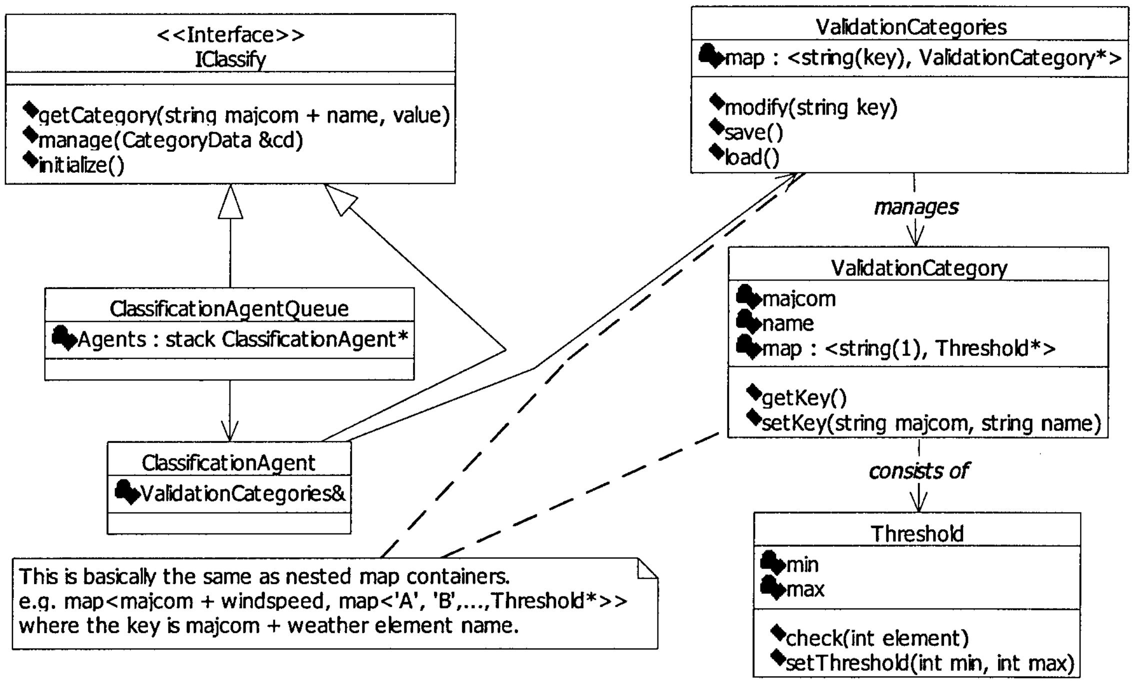


<<Interface>>
IPublication
publish(seq<ForecastData> &fs)
publish(seq<ObservationData> &os)
publish(seq<ValidationReport> &vr)
publish(seq<AlarmData> &ad)
initialize()
Instantiates the appropriate algorithm subclass with forecast data. Implements a "Strategy" pattern that encapsulates a family of algorithms, making each interchangeable.
PublicationAgentQueue
Agents : queue PublicationAgent*
PublicationAgent
<<Abstract>>
ReplicationAlgorithm
publish()
alarms, result data
Pipeline
PrimaryCopy
Event
<<CORBA Interface>>
IRCollection
(from TFMS Delegation Module)
add(in seq<ForecastData> fs)
add(in seq<ObservationData> os)
add(in seq<ValidationReport> vr)
manage(in StationData sd)
manage(in CategoryData sd)
view(in string query)
schedule(in string query, in time hour)
<<CORBA Interface>>
EventChannel
(from CORBA Event Service)
connect_consumer()
connect_supplier()

Figure 29: TFMS Publication Module

3.5.2.7 Classification Module

The TFMS classification module is shown in Figure 30. The purpose of the classification module is to categorize weather attributes IAW defined MAJCOM category thresholds and return this information to the caller.



<<Interface>>
IClassify
getCategory(string majcom + name, value)
manage(CategoryData &cd)
initialize()
ClassificationAgentQueue
Agents : stack ClassificationAgent*
ClassificationAgent
ValidationCategories&
ValidationCategories
map : <string(key), ValidationCategory*>
modify(string key)
save()
load()
manages
ValidationCategory
majcom
name
map : <string(1), Threshold*>
getKey()
setKey(string majcom, string name)
consists of
Threshold
min
max
check(int element)
setThreshold(int min, int max)
This is basically the same as nested map containers.
e.g. map<majcom + windspeed, map<'A', 'B',...,Threshold*>>
where the key is majcom + weather element name.

Figure 30: TFMS Classification Module

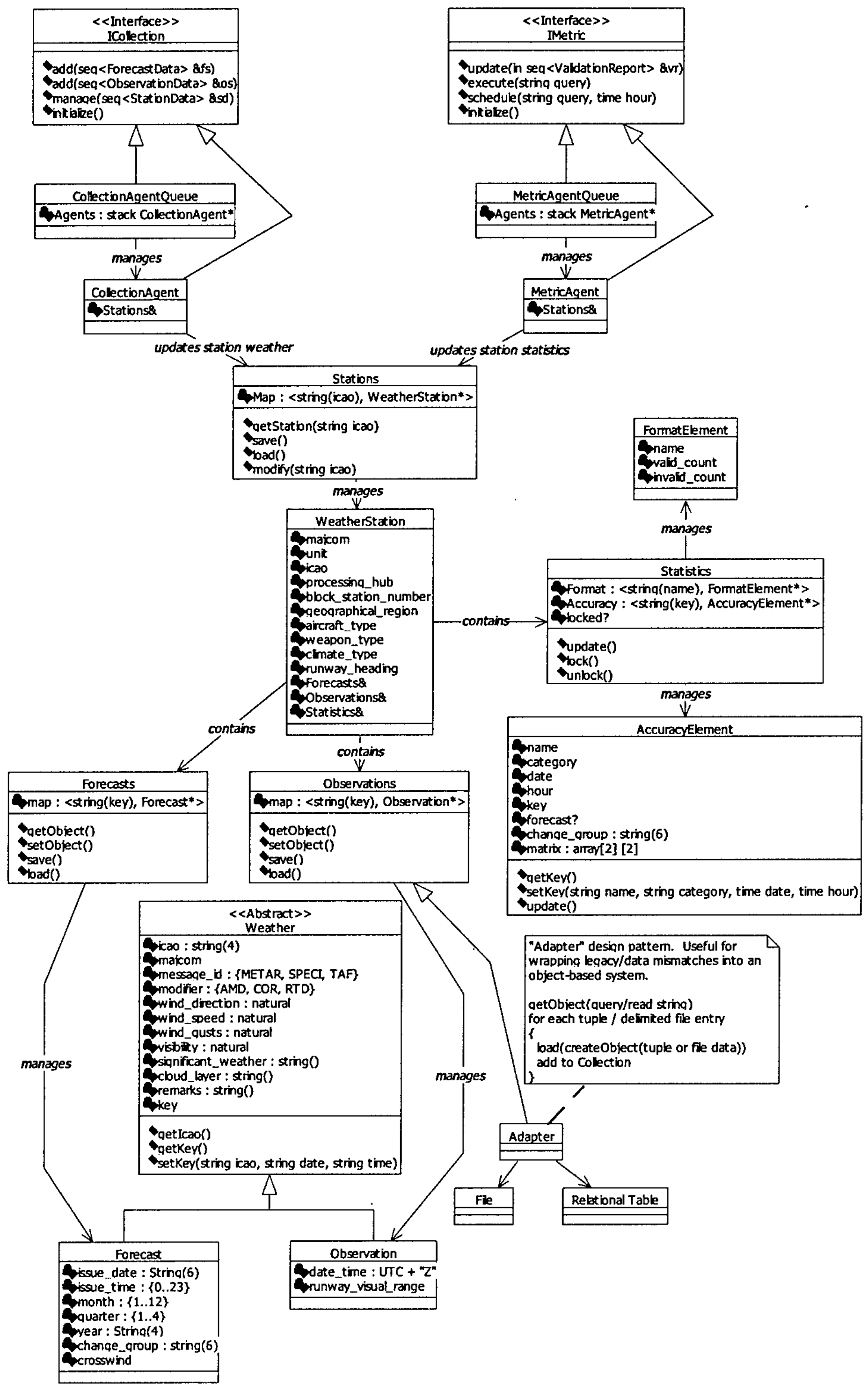
As shown in Figure 30, classification agents hold a reference to all defined categorical information and since category data is static, no data consistency issues exist with this module.

3.5.2.8 Data Module

Figure 31 shows the data module. The data module provides the necessary performance and availability required for TFMS customers to retrieve timely forecast, observation, and statistical information. The *delegation* module is reused to dispatch multithreaded data layer CORBA operations. Physical design is improved by removing all CORBA source file dependencies from the data module (§3.5.2.4 Delegation Module) [21 and 22]. When the data module initializes, forecast, observation, statistic, and weather station containers form a cache collection data object layer [32]. The TFMS schema is almost directly mapped from the software specification, e.g. a weather station contains a set of forecasts, observations, and statistics.

The collection and metric agents are mediators; controlling and coordinating object interaction to the cache collection [13]. The notification agent (§3.5.2.4 Delegation Module) is a colleague of the collection and metric agents, performing a *push* data delivery service between the CORBA Event Service and the *ICollection* and *IMetric* interfaces. In this way, colleagues are decoupled from each other and may be directly reused in other applications [13].

Since potentially many customers may require access to *read-only* forecast, observation, and statistic cache collections, search queries are a prime candidate for multithreaded application *and* load balanced distribution for increased availability if the collection module is replicated *within* an OWS. This functionality is easily added to the delegation module by using a *static* variable in the view command class to keep track of client connections (number of active queries). When a certain client threshold is reached, we can redirect the client using a *location agent* (not shown), which maintains handles to replicated collection modules on other local processing machines.



"Adapter" design pattern. Useful for wrapping legacy/data mismatches into an object-based system.

```
getObject(query/read string)
for each tuple / delimited file entry
{
  load(createObject(tuple or file data))
  add to Collection
}
```

Figure 31: TFMS Data Module

The adapter pattern *wraps* data stored as relational tuples or files into data layer objects in cache collections [41 and 43]. This pattern is only required if the underlying datastore is *not* an object-oriented database management system (OODBMS). By wrapping forecasts in this manner, we do introduce a performance penalty when building the object containers or flushing a given object from cache to disk. This penalty is commonly referred to as *impedance mismatch* and surfaces whenever we have to map one language to another [44].

The *IMetric* interface provides an update operation for bulk validation reports. Concurrency issues do exist with the format element and accuracy element containers. When the subsystem receives a set of TAF/observations, *all* validation elements must be updated. Once again, there are two ways to implement lock granularity: at the container level or at the validation element object level. In this case, updating the validation elements makes more sense at the container level, due to the query possibilities of the analyst. In other words, data consistency/dependency issues exist between validation elements, whereas in the trend object, there were no inter-object dependencies (§3.5.2.5 Validation Module). Accuracy elements use a unique key composed of four properties: 1) name, e.g. wind speed, 2) category, e.g. A, 3) date, e.g. 19991215, and 4) hour. Format elements use a name key, e.g. wind speed, as they simply track the number of valid and invalid entries for a given weather attribute.

3.5.3 System Performance, Evolution, and Reliability Considerations

System performance is determined by a number of factors (§4.2 Performance Benchmarks). Some issues such as impedance mismatch, processor speed, and network latency are unavoidable. Items such as using threads for computationally expensive operations, event loops for communication intensive operations, and pre-creation facilities such as the evictor design pattern to reduce system-processing delay are used where the technique seemed most appropriate.

Other design patterns are used primarily for easy system extension. The overloaded *execute* method of each derived command (or control) class carries out functions required by a specific CORBA operation (making all calls to the appropriate objects). In this way, each command object decouples objects that invoke operations from objects that implement the behavior, greatly adding to reuse of system objects. Also, it's very easy to add new command objects without recompiling the client since the delegation layer isolates the CORBA classes from the implementation language [6].

Parameterized types also aid system reuse and reliability as well. For the TFMS, the primary opportunity for reuse was container implementations. As mentioned above, the C++ STL provides many different data structures for the application developer, e.g. vector, list, stack, map, multi-map, etc. [9]. Using the C++ STL increases application reliability and software reuse while reducing coding and debugging time since its containers and algorithms are fully tested and debugged [51 and 70]. The map container was primarily chosen for order one lookup performance, since the keys were all unique, and the stack was chosen for its ability to easily determine unused system agents for better memory management facilities [21].

Implementing *very* course granularity does *generally* possess the same disadvantages as most centralized approaches: 1) single point of failure and 2) poor scalability [12]. In addition, adding or deleting *interface* operations may mean IDL, client, and server recompilation, so *comprehensive* interface definition is not only important, it's critical - even if operations aren't fully supported yet. The main disadvantage with fine granularity is increased communication within the distributed system, so there are certainly tradeoffs to consider, e.g. distributing *n* rules over *n* machines, as opposed to *n* rules on one machine.

3.6 Summary

In this chapter, an OO design methodology is employed to develop the TFMS. This methodology shows how the TFMS is partitioned into a distributed system and addresses the issues of forecast validation, publication, and metric processing functions. The OO lifecycle chosen is based on distributed OO data systems design and shows how TFMS requirements are developed and transformed across the lifecycle [27].

IV. Distributed System Prototyping

On the other hand, we cannot ignore efficiency. - Jon Bentley

4.1 Introduction

An important aspect of distributed system design involves performance measurement and determining the effectiveness of the design. This involves understanding performance benchmarks, the operating environment, and the application's functional requirements to determine if the software system effectively implements what the customer requested (§3.4.1.2 Functional Description). All factors affecting the experimental process are described, then a client/server measurement model is examined to uncover appropriate measurement points within the system. Experimental parameters are also defined along with system measurements.

4.2 Performance Benchmarks

A benchmark is a performance testing application that captures the data processing characteristics for a *class* of applications [10]. No single metric has the ability to convey computer system performance for all applications. System performance significantly varies from one problem domain to another and distributed object system performance is no exception. Domain-specific metrics specify a synthetic workload that characterize a typical application for a given problem domain [54]. The performance of this workload using various hardware platforms, distributed object system implementations, or alternative algorithms provides valuable insight into the relative performance of each alternative as it relates to the problem domain. For a benchmark to be useful, we must keep the following factors in mind [54]:

1. Relevance: Must measure performance relative to typical operations within the problem domain.
2. Portability: Easy to implement on different architectures.
3. Scalability: Should apply to small and large systems alike, i.e. the benchmark itself should be scaleable.
4. Simplicity: The benchmark must be understandable or it may lack credibility.

This section considers performance metrics used to evaluate and determine relative performance of the TFMS application. This research is applicable to a wide range of applications and is written with the intent of providing insight into the metric definition process, specifically addressing the measurements required or that should be considered for distributed object system development.

4.2.1 Basic Performance Metrics

Execution time and floating-point operations (FLOP) are frequently used to measure a program's computational workload.

4.2.1.1 Execution Time

Program execution time is the total time taken to execute a program and is measured as wallclock, response, or elapsed time [55]. Execution time depends upon many factors. Table 3 summarizes major factors affecting execution time.

Table 3: Factors Affecting Execution Time

Execution Time Factors	
Algorithm	An algorithm's asymptotic performance has a huge impact on execution time e.g. a bubble sort O (N^2) versus a quick sort O (N Log N).
Data Structure	How data is structured affects processing time e.g. a simple data type takes much less time to marshal than a complex constructed type like a nested structure [21].
Program Data	Certain programs are immune to input data values e.g. an N-point Fast Fourier transform, but others are greatly affected by input data values, e.g. comparison-based sorts such as quicksort, whose best or worst case time greatly depends upon the partitioning of the input sequence [15].
Operating Platform	Machine/Instruction set architecture, operating system and version, compiler and version, memory hierarchy (multiple cache level, memory, and disk bandwidth), and application use (dedicated or timeshare) all affect execution time [10 and 55].
Programming Language	Low-level or high-level languages, compiler optimizations, and code optimizations can greatly affect execution time e.g. In C++, *inlining* friend and member functions, and removing unneeded destructor and copy constructor/assignment definitions greatly decreases execution time [52].
Interconnection Network	Network characteristics (shared/switched, LAN/MAN/WAN), network bandwidth, communication latency, and delays caused by buffering interconnection device mismatches e.g. high-speed to low-speed and fragmentation/reassembly operations all greatly affect a program's execution time [10 and 11].

4.2.1.2 Floating-Point Operations

In applications where numerical calculations dominate, a metric frequently used is floating-point operations. The unit of measurement is the number of floating-point operations per second (FLOPS). For a meaningful metric, rules must be followed for counting floating-point operations [10]:

Table 4: Some FLOP Counting Rules

Rule	Number of Floating-point Operations	Comment
X = 1.2 + 2.4 * 1.22 - 1.11;	3	Addition, multiplication, and subtraction count as one FLOP each.
X = Y;	1	Isolated assignment.
If (X > Y) X * 2.2;	2	Comparison counted as one FLOP.
X = (float) Y * 2.0;	2	Typecast counted as one FLOP.
X = Z / 3.21 * sqrt (Y);	9	Division or square root counted as four FLOPS.
X = sin (Y) * exp (Z);	17	Sine, exponential, etc. counted as eight FLOPS.

4.2.1.3 Combinations of Basic Metrics

Execution time and floating-point operations can also be used in a meaningful way when benchmarking different design alternatives, e.g. testing if multithreading or event-based server design is more suitable for a given computational load (§4.2.2.3 Server-Side Performance). In this report, a benchmarking methodology is implemented where a certain FLOP workload is applied to a server, and response time is measured to determine the computational workload where a threaded approach may be more appropriate than an event-based server implementation for a given number of clients.

4.2.2 CORBA Performance Metrics

In distributed computing, whether you're designing business applications using distributed objects or allocating processors to solve a computationally-expensive numerical problem, you must know the communication costs associated with the application. This means you must know the cost of sending remote messages, which is determined by two factors: latency and marshaling rate. One point to always remember is that results may vary widely depending on the operating environment used for measurements. For measurements to be meaningful, developers should always create a *prototype* of the deployment environment to mitigate all confounding factors [49].

4.2.2.1 Call Latency

Call latency is the *minimum* communication cost you incur when invoking a remote operation. The cheapest CORBA message is one that has no parameters and returns no result using the CORBA IDL *oneway* keyword. One-way CORBA operations use *best-effort* delivery semantics and must follow three rules: 1) must have a *void* return type, 2) must not contain *out* or *inout* parameters, and 3) must not contain a *"raises"* expression [21].

Measuring call latency determines a fundamental design limit for the ORB implementation you're comparing or using. If the application requires more remote message invocations than the ORB is capable of delivering, either a better performing ORB must be found or the application must be optimized to meet this limitation. For example, suppose you measure the call latency for two commercial vendor ORBs. One vendor's ORB has a call latency of 1 millisecond (ms) and the other vendor's ORB a 2 ms call latency. Your application is optimized, but still requires 600 remote calls per second (sec). Considering the two ORBs above, you can expect a maximum call dispatch rate of 1000-calls/sec for one ORB implementation, while the other delivers a 500-calls/sec maximum dispatch rate. Based on the call latency benchmark, you'd most likely choose the 1000-calls/sec ORB implementation for your distributed object system.

Another important consideration is hardware utilization. We gain insight into hardware utilization by determining the *efficiency* of a particular operation, where efficiency is a measure of the fraction of time that a processor is usefully employed [15]. Mathematically, efficiency is defined by [15]:

$$E = S/p$$

Where S is the *speedup* ratio of serial run time to parallel run time and p is the number of processors.

In a client/server model (§4.3.3 Client/Server Measurement Model), the number of processors is one and efficiency effectively becomes equal to the speedup ratio. If efficiency is a design issue, we can determine the amount of server work needed to obtain a target efficiency by using the call latency benchmark and measuring the time for the cheapest *local* procedure call (LPC), i.e. local latency. In this way, we can use the standard speedup equation to determine where to place certain operations within the distributed system. For example, consider our cheapest local operation costs 1 microsecond, while our ORB's call latency costs 1 msec. From the standard speedup equation, $S = 0.001$, so our remote calls are 1000 times slower. If target efficiency is 30 percent or above, then the operation workload must roughly exceed 0.5 msec to properly utilize the remote machine and mitigate communication costs (~0.5msec/0.5msec + 1msec = ~33% efficiency).

4.2.2.2 Marshaling Rate

The marshaling rate is the speed in which an ORB can transmit and receive data over the network. Marshaling performance depends upon the data type being transmitted over the network [21]. Simple types typically marshal fastest because they're usually a simple block copy into the ORB's transmit buffer. Marshaling complex data types and object references usually takes much more time because the ORB must do more work at run time collecting data from different memory locations, and then copying this data into the transmit buffer.

When determining an application's marshaling rate, it's important to use the *actual* data types used by the application in its *typical* operating environment. Designers should determine the time to marshal, unmarshal, dispatch, and service each remote call for future use in performance calculations and measurements [49].

4.2.2.3 Server-Side Performance

Because CORBA is server-centric, most scalability and performance improvements are realized on the server-side [21]. Multithreading is certainly at the top of a server developer's to-do list, but event-based approaches can also be used as an alternative to multithreaded server applications. In this section, the two approaches are briefly discussed.

Multithreaded server applications can provide many benefits like simplified program design, improved throughput, true concurrency on multiprocessor hardware platforms, and improved response time enforced by the operating system scheduler [21 and 56]. To realize these benefits though, multithreading is *practically* applied toward four areas: 1) offloading time-consuming tasks from the main thread, 2) providing application scalability for SMP systems, 3) sharing computing resources fairly, and 4) driving independent players in simulations [50]. Server applications are supposed to operate efficiently during peak usage periods, when many active clients require simultaneous service. Given this environment, a single-threaded implementation would make a poor choice if certain requests require disk or network I/O, since these requests would block the server application for extended periods of time, starving other client requests [11, 12, and 56]. If the server application only processes short-duration requests though, a single-threaded implementation may be the right choice to minimize application complexity [47].

Multithreaded ORB implementations have a wide variety of options available to handle requests. Table 5 provides a summary of the most popular ORB thread architectures. Once the ORB core dispatches a client request to a portable object adapter (POA) where the target object is located, the POA threading policy takes over [21]. A POA can have a SINGLE_THREAD_MODEL or ORB_CTRL_MODEL value for its thread policy. When a POA has the SINGLE_THREAD_MODEL value set, all servant invocations are serialized. For a multithreaded ORB, when a POA has the ORB_CTRL_MODEL value set, this implies that a POA is *allowed* to dispatch multiple requests concurrently, but it says nothing about the thread model used to handle these requests [21]. As Table 5 shows, there are advantages and disadvantages with each thread model, so application requirements, client access patterns, the operating environment, and performance testing become important factors when choosing a vendor's ORB implementation.

Table 5: ORB Multithreading Architecture Summary [56]

Architecture	Comments	Advantages	Disadvantages
Thread-Per-Request	Handles each client request as a separate thread of control.	Straightforward to implement. Useful for long-duration requests.	Consumes many OS resources for many clients. Inefficient for short-duration workloads – expensive thread creation costs.
Thread-Per-Connection	Variation of thread per request. Each client connection has a separate thread of control.	Straightforward to implement. Useful for long-duration requests.	Does not load balance effectively. Can over-utilize connection.
Thread-Per-Servant	Associates a thread with each servant (object instance).	Useful to minimize rework of single-threaded servants.	Does not load balance effectively. Can over-utilize servant.
Thread-Pool	Variation of thread per request.	Amortizes thread creation costs.	Must dedicate resources.

Event-based servers are a design approach that offers many advantages: 1) no complicated resource locking, 2) no data corruption, and 3) minimal overhead [47]. Replacing all blocking I/O with event notification and distributed callback mechanisms increases server performance for *short* duration events, but event-based servers quickly lose these performance advantages when events take too long to process [47 and 50]. Event-based services are appropriate in distributed environments where communication-oriented tasks such as web services or transferring small amounts of data are prevalent. Measuring an ORB's Event Loop throughput and comparing to a suitable multithreaded implementation would greatly aid our decision-making process for implementing server-side functionality for a particular operation.

4.3 Experimental Design

One point previously mentioned (§4.2.2 CORBA Performance Metrics), concerns the issue that an experimental result may vary widely depending on the operating environment (§5.3.2 Experimental Environment) and deployment model (§5.3.1 N-Tier Topologies) used for measurements. For measurements to be meaningful, these conditions must be recorded, e.g. a processor's clock rate alone inversely affects the machine's execution time since CPU Execution Time = Program Clock Cycles ÷ Clock Rate [55].

4.3.1 Factors

To obtain unbiased results, the following factors are recorded for reproducibility purposes [45]. These factors can greatly impact the anticipated results from experimental reproduction or could possibly invalidate experimental data. Every attempt to reduce or eliminate the impact of these factors (§4.3.2 Mitigation of Factors) is performed to ensure experimental integrity.

1. High Resolution Global Time: The clocks on two different machines may not be correlated to a centralized time source. If global time is not achieved, certain measurements cannot be accurately performed.
2. Machine Utilization: Business applications typically share processor time with other applications, i.e. applications use timesharing.
3. Network Utilization: While applications typically share processor time with other applications, machines must share network bandwidth with other machines.
4. LAN/WAN Interface Buffering: These experiments are performed using a 100 Mbps Fast Ethernet internetwork. The overhead in processing data packets for different communication protocols and rates is not considered.
5. Data Security Implementation: These experiments are performed with **no** security methods. The overhead in processing secure data packets along with any encryption mechanisms is not considered.

6. MICO IDL Compiler: MICO's IDL compiler and library support improves with each new release, and future versions may greatly affect the time to marshal and unmarshal data. Other IDL compilers may contain optimizations, which may decrease data packing and unpacking time as well.
7. Measurement Overhead: Obviously, a certain amount of overhead is incurred with the measurement activity.
8. Time Constraints: This research report has a deadline. There is not enough time to try every possible optimization and measure/analyze the results.
9. Software Portability: The code should run with minimal effort on different operating platforms, e.g. all flavors of Unix (Linux, Sun, Solaris, etc.), mainly for reproduction of experiments and platform comparison purposes [48]. Sometimes, writing portable code may sacrifice program performance.

4.3.2 Mitigation of Factors

Mitigation of factors (§4.3.1 Factors) attempts to improve the documentation of the experiments and reduces or eliminates the effects of each factor, greatly contributing to overall experiment validity.

1. High Resolution Global Time: All machines are members of the same Windows 2000 domain model. Immediately before experiment execution, machines will synchronize their clocks with the Domain Controller (DC).

2. Machine Utilization: Machine utilization is checked prior to experiment execution. Only measurement applications, e.g. Windows 2000 performance monitor, are running besides the experiment.
3. Network Utilization: Network utilization is checked prior to experiment execution. Every attempt is made to use the operating environment while network utilization is minimal.
4. MICO IDL Compiler: MICO's IDL compiler is the latest version.
5. Measurement Overhead: The wallclock time required to take a time measurement is recorded and subtracted from the experiment's time.
6. Software Portability: All CORBA code is written to the CORBA 2.3 specification with no proprietary hooks. Platform-specific Win32 API thread functionality is encapsulated in a single *ThreadPool* class (§5.4.1.3.1 Threaded), which isolates platform modifications to a single class.

4.3.3 Client/Server Measurement Model

Building a client/server measurement model to measure remote operation costs is a fairly straightforward process. Figure 32 shows experimental measurement points we could use in a typical client/server interaction. For each experiment, a table like Table 6 is used to record measurement data. For example, let $T_{marshal}$ be the time to marshal a particular data type on the client. To compute this time, we would use $T_{marshal} = T_2 - T_1$, where T_2 is the time at measurement point two and T_1 is the time at measurement point one.

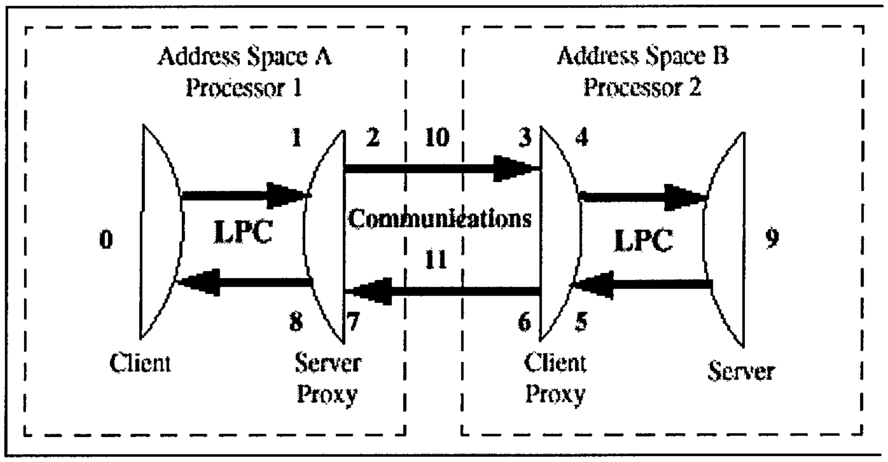


Figure 32: Client/Server Measurement Model [49]

Table 6: Remote Operation Costs

Measurement Point	Comment	Operation 1, ...	..., Operation N
0	Client Execution time		
1	Before client marshal		
2	After client marshal		
3	Before server unmarshal		
4	After server unmarshal		
5	Before server marshal		
6	After server marshal		
7	Before client unmarshal		
8	After client unmarshal		
9	Server Execution time		
10	Bytes transferred to server		
11	Bytes transferred to client		
Network Utilization (Point 10)	Bandwidth Used		
Network Utilization (Point 11)	Bandwidth Used		
Total Operation Time			
Total Bytes Transferred			

4.3.4 Parameters

Table 7 presents the parameters and values used to conduct TFMS experiments. Each parameter is described below:

1. Data Type: Three data types are used in the experiments. A simple double type, a Forecast structure, and a sequence of Forecasts are used to measure marshaling rate and normal versus bulk operation experiments.
2. Number of Forecasts: A range of one to 1000 forecasts is used in the replication and validation experiments. Bulk compared to normal operation performance is considered as well.
3. FLOP (§4.2.1.2 Floating-Point Operations): Workload applied to a single function or transaction. Workload ranges from 1000 to 10,000,000 FLOPS.
4. Number of clients: A range of one to five clients is used for testing server-side performance. A client may be a thread or lightweight process.
5. Number of business/collection servers: A range of one to seven servers is used to test both small and large distributed configurations.

Table 7: Experiment Parameters

Parameter	Values
Data Type	double, forecast, sequence<forecast>
Number of Forecasts	1, 10, 100, 200, 300, 400, 500, 1000
FLOPS	$1*10^3$, $10*10^3$, $100*10^3$, $1*10^6$, $10*10^6$
Number of Clients	1, 2, 3, 4, 5
Number of Servers	1, 2, 3, 4, 5, 6, 7
Replication Algorithm	CORBA Event Service, Pipeline, Primary Copy
Operation	Normal, Bulk

6. Replication Algorithm: CORBA Event Service, Pipeline, and Primary Copy algorithm execution times are compared to determine which is most suitable for the TFMS.

7. Operation: Normal versus bulk operations is compared to determine which is most appropriate for a given transaction.

4.3.5 Measurements

The experiments performed in this report follow a specially constructed test problem methodology [45]. The following metrics are used for the TFMS experiments:

1. Execution Time (§4.2.1.1 Execution Time): The time elapsed between the beginning of the program to the last node's program execution [15].
2. FLOPS (§4.2.1.2 Floating-Point Operations): The number of floating-point operations executed by a single function or transaction per unit time.
3. Throughput (§4.2.1.3 Combinations of Basic Metrics): Number of jobs (FLOPS, forecasts, or transactions) processed per unit time [10].
4. Network Utilization: Amount of bandwidth used as a percent of total bandwidth when executing inter-site experiments (Replication).

4.3.6 Measurement Confidence Level

To determine the number of measurements required for each experiment, ten sample measurements are taken to determine a mean and standard deviation. Once this information is computed, these values are used to determine the number of measurements required for a confidence level of 96 percent possessing an accuracy of one percent from the mean. Confidence limits for a population mean are given by [61]:

$$X \pm (Z_c * S)/n^{1/2}$$

Where X is the sample mean, Z_c is the confidence level, S is the sample's standard deviation, and n is the number of samples. The quantity $(Z_c * S)/n^{1/2}$ is also equal to the mean times the required accuracy p, so this becomes:

$$X * p = (Z_c * S)/n^{1/2}$$

Therefore, the number of measurements required for an experiment can be found using the following formula ($Z_c = 2.05$ for 96% confidence level [61]):

$$n = [(2.05 * S)/(X * 0.01)]^2$$

4.4 Summary

An important aspect of distributed system design involves performance measurement and determining the effectiveness of the design. This chapter discusses basic and distributed system performance benchmarks, while also addressing experimental design of the TFMS. All factors affecting the TFMS experimental process are outlined. A client/server measurement model is examined to uncover appropriate measurement points within the TFMS system. Finally all experimental parameters and measurements are defined outlined.

V. Distributed System Implementation

Programming is understanding. - Kristen Nygaard

5.1 Introduction

In the previous chapter, a design methodology was used to map business requirements to a software system. Software system implementation and experimentation details are presented in this chapter. The implementation phase is used to map the software system to an operating environment. TFMS experiment implementation details, experimental operating environment, and transaction characteristics are explained and documented. Various distributed topologies along with their inherent component placement tradeoffs are also examined. As previously noted (§1.7 Summary), the TFMS is a prototype CORBA application developed to quantify the impact of specific design decisions in a distributed environment.

5.2 Software Development Environment

The development tools used in this research are the MICO CORBA 2.3-compliant C++ implementation and Microsoft Visual Studio 6.0 Enterprise Edition. The primary reasons for choosing these tools is convenience, familiarity, and lack of time to pursue other CORBA implementations (§4.3.1 Factors). Many CORBA vendors and CORBA-aware CASE tools exist. Most of these resources are available on the WWW [77].

5.2.1 MICO

MICO (stands for *M*ICO *i*s *CO*RBA) is a *freely* available and fully compliant C++ implementation of CORBA version 2.3 [34]. MICO is compatible with a great number of operating platforms, using the native C++ compiler in most cases. To implement a CORBA application, CORBA IDL is specified and compiled using the MICO IDL compiler. MICO generates the C++ mapping in the form of header and body files containing all the necessary *CORBA* code required for application implementation. Application developers must *implement* the virtual method definitions in these generated files to bind the CORBA interface definitions to C++ objects.

5.2.2 Microsoft Visual Studio 6.0 - Enterprise Edition

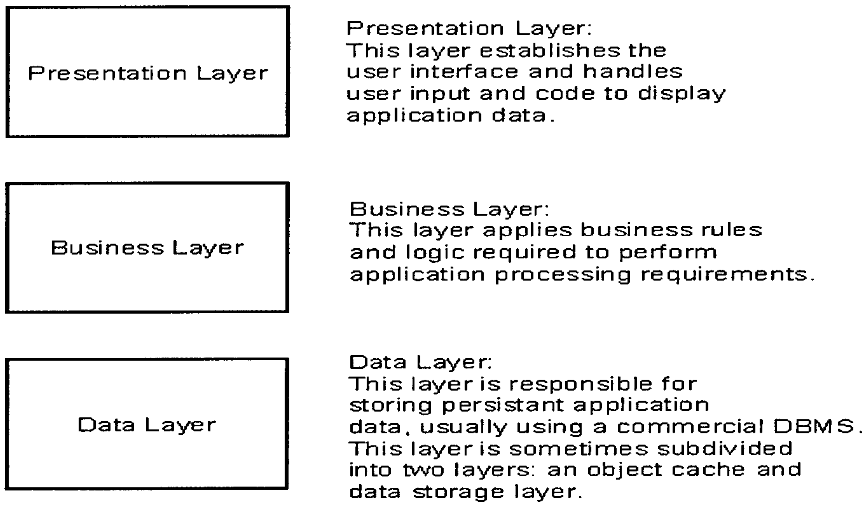
Microsoft Visual Studio 6.0 Enterprise edition includes *Visual Modeler*, a graphical OO design tool, and *Visual* C++, Microsoft's C++ development environment. Once the MICO IDL compiler compiles CORBA IDL into C++ header and body files, MICO libraries are configured in the VC++ development environment and application development continues as a normal VC++ project [34]. No compiler optimizations are applied to these experiments. Visual Modeler is used to create the TFMS object model and is the source of several figures shown in the report. Visual modeler is also used to generate *skeleton* code (class headers, constructors, method declarations, etc.) for C++ classes.

5.3 Deployment

Deployment maps the software architecture to a specific operating environment. Within the operating environment, there usually exists a variety of ways to configure distributed components.

5.3.1 N-Tier Topologies

When deploying a distributed application, there are many ways to arrange application components onto physical machines. The logical N-tier model shown in Figure 33 may be arranged in many different ways depending on application and performance requirements.



Presentation Layer:
This layer establishes the user interface and handles user input and code to display application data.

Business Layer:
This layer applies business rules and logic required to perform application processing requirements.

Data Layer:
This layer is responsible for storing persistant application data, usually using a commercial DBMS. This layer is sometimes subdivided into two layers: an object cache and data storage layer.

Figure 33: N-Tier Logical Model

In this section, five physical implementations are discussed: 1) Two-Tier implementations with *thick* clients, 2) Two-Tier implementations with thick servers, 3) Three-Tier implementations with *thin* clients, 4) Three-Tier implementations with thick clients, and 5) N-Tier implementations.

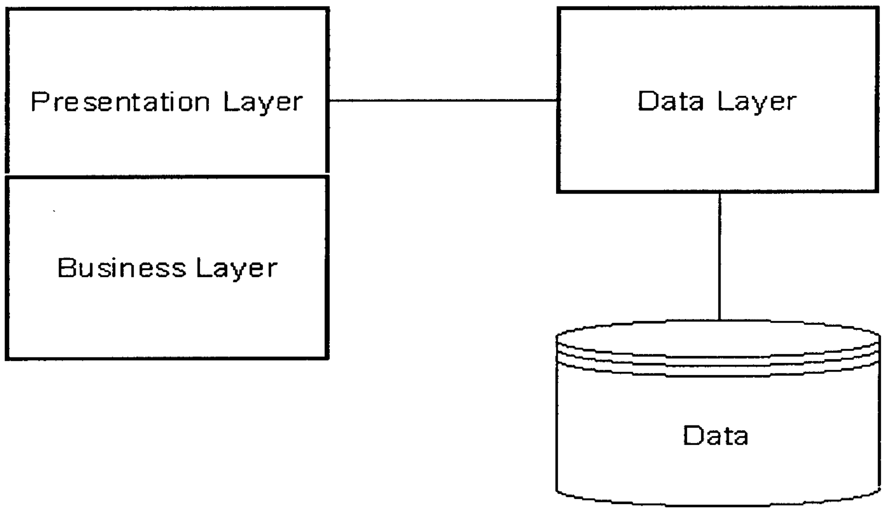


Figure 34: Two-Tier Thick Client

A common method for deploying a client/server application is shown in Figure 34. This two-tier implementation uses thick [3] clients, where all business and presentation logic is physically executed on client machines. In this implementation, the server acts as a traditional database server. A primary advantage with the thick client implementation is that user interface tools supporting this model are powerful and well established; e.g. Visual Basic is in its sixth major version [62]. A disadvantage with this implementation is that deploying all business logic on the client generally means more communication overhead because data must be moved to the client to apply business logic. Another disadvantage is that changes to business logic require recompilation of all clients.

[3] The term "FAT" or "Thick" generally refers to what system entity possesses the majority of the application logic.

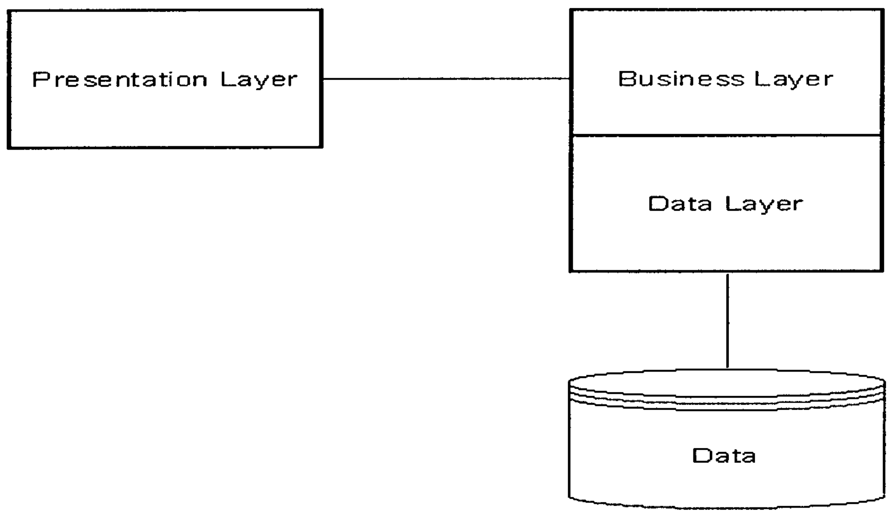


Figure 35: Two-Tier Thick Server

In a two-tier thick server implementation, business logic is physically executed on the server and is generally written as stored procedures and triggers within the database. For example, in the TPC-C benchmarks published for Microsoft SQL Server, the core transaction logic is coded as stored procedures in the server [2]. The major advantage of a thick server implementation is performance. Business logic runs in the same address space as database code and is tightly integrated with the database search engine, so data is not moved or copied before it's operated on, therefore minimizing network traffic. The main disadvantage of this implementation concerns scalability - application scalability is tied to the hardware platform where the server resides [7 and 8].

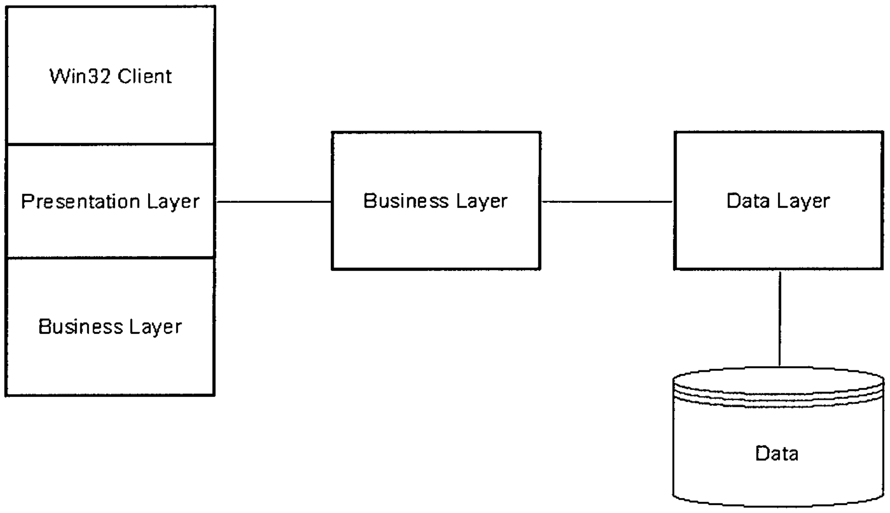


Figure 36: Three-Tier Thick Client

The key distinction of a three-tier implementation is the existence of a processing boundary; between the data, business, and presentation layers; this processing boundary may be physical or logical, i.e. different physical machines or different address spaces on the same physical machine [7]. Transaction processing and object transaction monitors (TPM and OTM respectively) such as Microsoft's MTS and BEA System's Tuxedo/M3 products use this topology to provide process, transaction, communications, and load balancing services for large transaction-oriented, e.g. financial, distributed environments [7].

The three-tier thick client implementation in Figure 36 shows business logic residing in a client address space, while other business logic resides in a separate address space on a different physical machine. If the business logic is computationally expensive, requiring a lot of processor time or physical memory, it's usually advantageous to locate these functions on one or more physically separate machines to minimize resource contention, increase processing efficiency, and decrease overall application execution time [15]. The potential scalability gain may be offset by the additional communication cost involved with moving data to middle-tier machines, so care must be taken when physically partitioning the application (§4.2.2 CORBA Performance Metrics). Another dimension of application scalability is added by three-tier applications accessing distributed databases, i.e. databases physically partitioned across multiple machines (§2.5.1 Architectures). Partitioning databases in this manner, however, introduces enormous complexities into the application and is not a widespread practice in industry today [62].

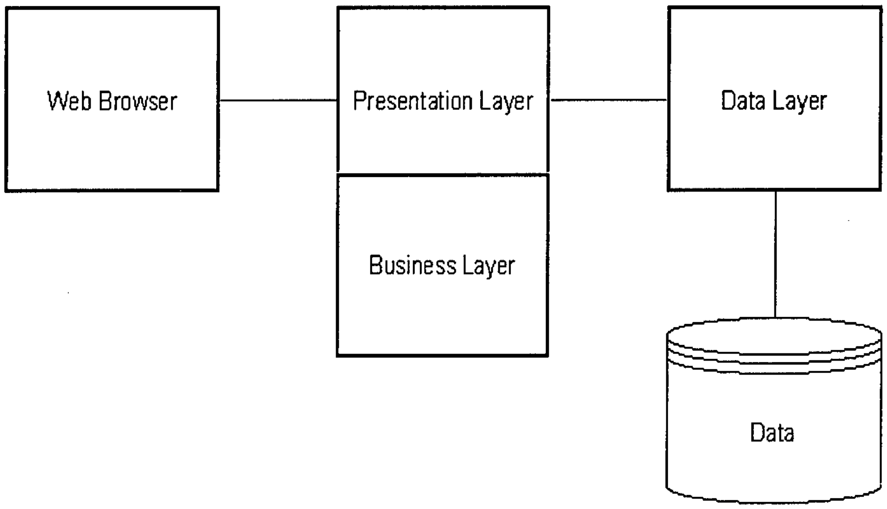


Figure 37: Three-Tier Thin Client

Figure 37 shows the topology for a three-tier thin client, i.e. Internet implementation. Internet implementations typically execute both business and presentation services using a Web server in the middle layer. Some products, e.g. WebLogic's Tengah architecture, execute the business logic on the Web server as well, i.e. in the same address space, thus avoiding the additional call overhead associated with crossing an additional process boundary [63]. One key advantage of Internet implementations is that anybody with a browser can access these applications. Standard Web browsers provide all required client functionality. Application evolution and upgrades are easily managed as well, since an update to the Web server automatically updates all clients. This is in sharp contrast to managing thick clients, where upgrades may affect application code at many [4] clients.

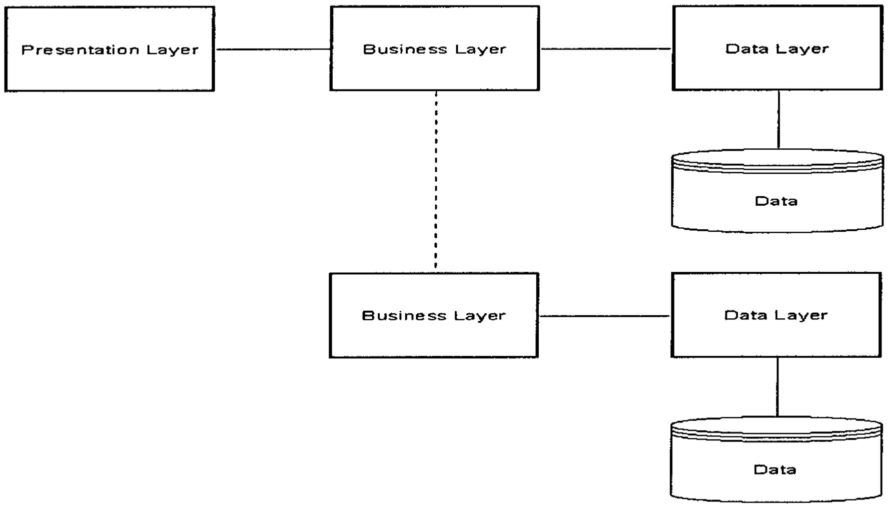


Figure 38: N-Tier Component Model

Figure 38 shows a topology where we lose the distinction of numbering the layers into a three-tier processing environment. In the N-tier topology, we start using white page (OMG Naming service), yellow page (OMG Trader and Query Services) directories, and smart component (agent) interactions as the basis for our application's behavior [8 and 14]. The same advantages/disadvantages discussed above exist with the N-tier topology as well, but

[4] The use of the word "many" here denotes the typical Internet user population.

with N-tier distributed applications, the *component* is the primary autonomous entity in the system [17].

5.3.2 Experimental Environment

The AFIT Bimodal cluster consists of 21 PCs: four PII-333MHz, seven PII-400Mhz, one PII-450MHz, eight PIII-600MHz, and one dual PIII-550MHz symmetrical multiprocessors (SMP). All hardware platforms possess at least 128 MB of memory, 512K level one cache, and 100Mbps *PCI* network interface cards. The Pile of PCs (PPCs) can boot Red Hat 6.0 Linux or Microsoft Windows 2000 (bimodal), and both platforms support CORBA-based application development using MICO [34]. Windows 2000, beta release 3 is used for all experiments. The cluster interconnection network currently uses two 100Mbps, 24-port Intel Express Fast Ethernet switches uplinked to a six-port Intel Gigabit switch.

5.3.3 Component Implementation

This section describes component implementation for TFMS experimentation. Functionality and interaction for the TFMS name server, test generator, TFMS business servers, and TFMS collection servers are described.

5.3.3.1 CORBA Name Server

A CORBA Name server provides a mapping of server names to object references, similar to the Internet Domain Name Service, which maps Internet domain names to IP addresses [21]. A *name binding* is a name-to-reference association and a *name context* is an associative object that stores name bindings, typically implemented as a lookup table. The CORBA Naming Service is used in this research to advertise business and collection servers, solving the problem of how TFMS components get object references at runtime. Figure 39 shows a typical CORBA lookup operation the test generator performs to obtain a TFMS business server reference. A TFMS business server obtains collection server handles using the same sequence of operations.

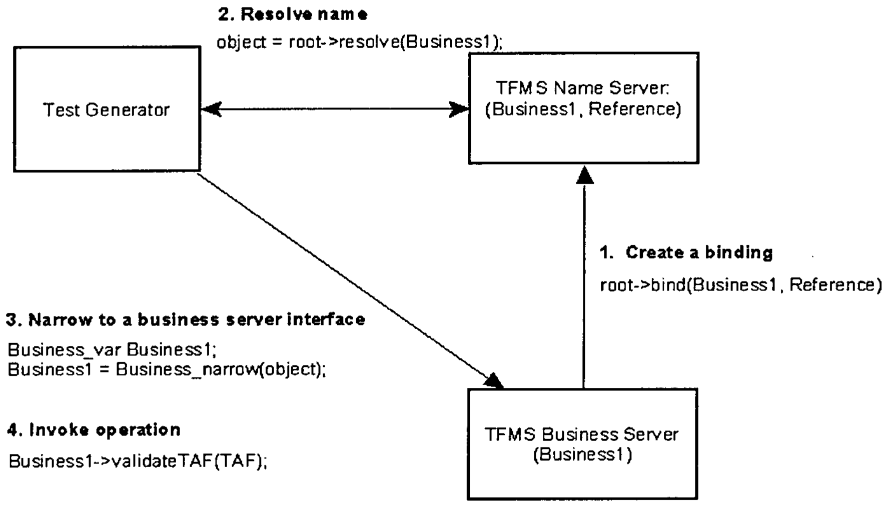

2. Resolve name
object = root->resolve(Business1);
Test Generator
TFMS Name Server:
(Business1, Reference)
1. Create a binding
root->bind(Business1, Reference)
3. Narrow to a business server interface
Business_var Business1;
Business1 = Business_narrow(object);
4. Invoke operation
Business1->validateTAF(TAF);
TFMS Business Server
(Business1)

Figure 39: Resolving TFMS Component Names

5.3.3.2 Test Generator

The test generator simulates multiple clients in the TFMS, offering the experimenter various options for testing different functions, e.g. ORB benchmarks or publication algorithm experiments in the distributed system. The test generator invokes CORBA operations on business and collection servers after obtaining interface references from the TFMS Name Server.

5.3.3.3 Business Server

The business server implements TFMS middle-tier services needed for TAF validation and publication. Business servers add themselves to the distributed system by registering their interfaces with the TFMS name server (§5.3.3.1 CORBA Name Server). All forecasts submitted for validation and publication use the TFMS business server. Business servers invoke CORBA operations on either business (validation function distribution) or collection (publication experiments) servers by obtaining interface references during runtime from the TFMS Name Server.

5.3.3.4 Collection Server

The collection server implements the data object layer associated with the data tier of the TFMS N-tier topology. The collection server maintains cache collections needed to provide high availability and performance [12, 41, and 43]. Collection servers add themselves to the distributed system by registering their interfaces with the TFMS name server (§5.3.3.1 CORBA Name Server).

5.3.4 Component Deployment

Figure 40 shows *one* TFMS deployment using the AFIT PPCs (§5.3.2 Experimental Environment). This deployment maps the test generator, name, business, and collection servers onto four distinct nodes (separate physical machines) running the Microsoft Windows 2000 operating system. A number of different test configurations are executed by *deploying* the appropriate number of business and collection servers, then selecting the appropriate test option from the test generator menu. This methodology is a convenient way to enable reproducibility or to perform cross-platform comparison [48].

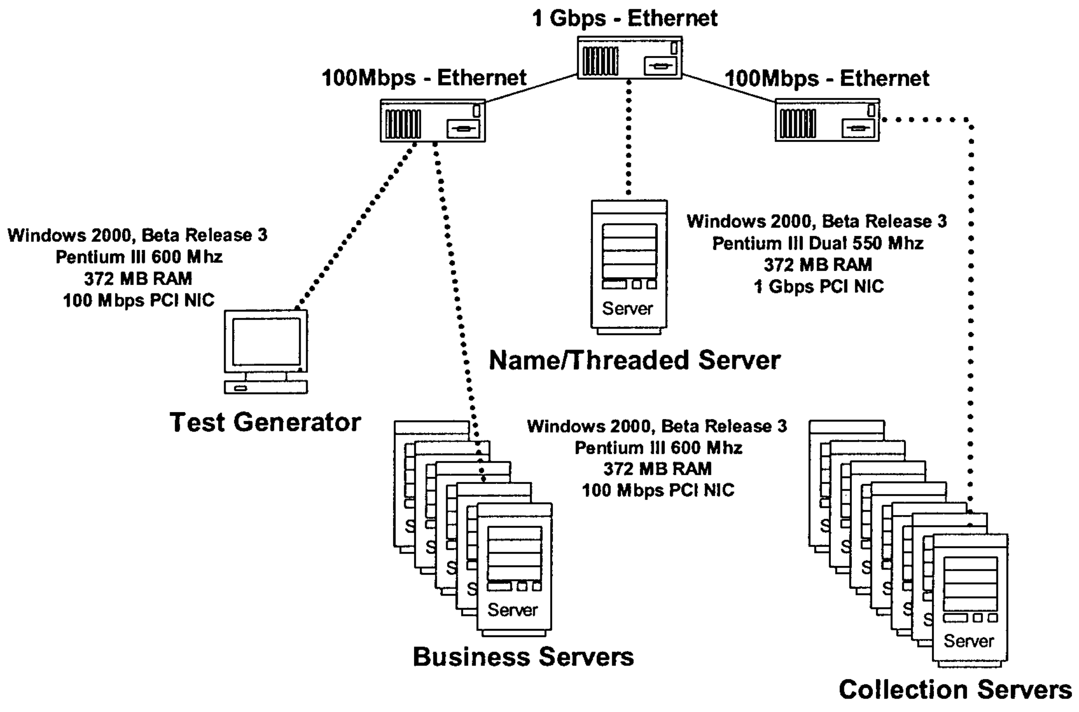


Figure 40: TFMS Deployment Diagram

5.4 Experiments

This section addresses the objectives of each experiment and the steps taken to implement the experiment. Data collection for each experiment is implemented by using C++ STL containers and algorithms encapsulated in an *Average* class. The Average class implementation is shown in Figure 41. Clients are required to collect measurements in a STL vector container. The *collect* function takes a reference to the client's measurement vector and a file name. Measurements are summed using the STL accumulate algorithm and divided by the size of the times vector to calculate the average. The sum, number of measurements, average, and all raw data are then saved to the file name provided by the client. All experiments in this research gather measurements non-intrusively in memory and then use this methodology to save experimental results. The number of iterations for experiment execution is determined using sample means and standard deviations from pilot transactions (§4.3.6 Measurement Confidence Level). Although some transactions required far fewer iterations, the minimum number of iterations for all experiments is the worst-case value of 100.

```
#include <numeric>
#include <fstream>
#include "Average.h"

Average::Average() {}

void Average::collect(vector<double> &d, char* s)
{
        times = d;
        ofstream outfile(s, ios_base::app);

        // Accumulate results
        double sum = accumulate(times.begin(), times.end(), 0.0);

        // Compute average
        double average = sum/times.size();

        // Save to file
        outfile.setf(ios_base::fixed, ios_base::floatfield);
        outfile << "Sum: " << sum << endl;
        outfile << "Measurements: " << times.size() << endl;
        outfile.setf(ios_base::fixed, ios_base::floatfield);
        outfile << "Average: " << average << endl;
                outfile.setf(ios_base::fixed, ios_base::floatfield);
        outfile << "Data: ";
        typedef vector<double>::const_iterator T;
        for (T t=times.begin(); t!=times.end(); ++t)
        {
                outfile.setf(ios_base::fixed, ios_base::floatfield);
                outfile << *t << ",";// dereference value at iterator t
        }
        outfile << "\n" << endl;
}
```

Figure 41: Average Class

5.4.1 ORB Benchmark Transactions

The objective of the ORB benchmark experiments is to determine a particular ORB's design limitations to aid the distributed object system engineering process (§4.2.2 CORBA Performance Metrics). Four ORB benchmarks are implemented: call latency, marshaling rate, server-side performance, and name server performance. Figure 42 graphically depicts ORB benchmark transactions.

5.4.1.1 Call Latency

The objective of this experiment is to determine the cost of a MICO ORB remote and local call. These measurements are required to compute relative efficiency for component placement within the distributed system (§4.2.2.1 Call Latency).

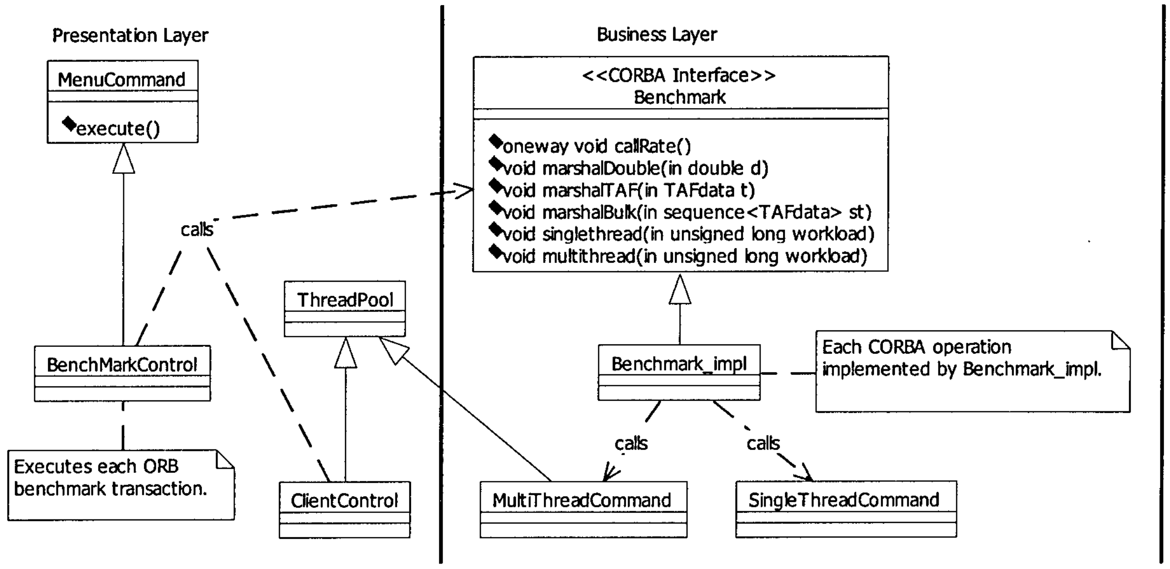

Presentation Layer
Business Layer
MenuCommand
execute()
<<CORBA Interface>>
Benchmark
oneway void callRate()
void marshalDouble(in double d)
void marshalTAF(in TAFdata t)
void marshalBulk(in sequence<TAFdata> st)
void singlethread(in unsigned long workload)
void multithread(in unsigned long workload)
calls
ThreadPool
BenchMarkControl
Benchmark_impl
Each CORBA operation implemented by Benchmark_impl.
Executes each ORB benchmark transaction.
calls
calls
ClientControl
MultiThreadCommand
SingleThreadCommand

Figure 42: Call, Marshaling Rate, and Server Performance Transactions

The call latency transaction is implemented using the following steps:

1. Prerequisites: Two machines are chosen for this test. The machine's clocks are synchronized with the Windows 2000 Domain Controller (DC). One machine hosts the benchmark server process and a remote machine hosts the test generator. The benchmark server binds and the test generator resolves interface handles using the TFMS name server upon startup.
2. The experimenter selects the ORB benchmark test when prompted by the test generator.
3. As shown in Figure 42, the test generator instantiates a *BenchmarkControl* object to control experiment execution.
4. The *BenchmarkControl* object records a transaction start time, and then calls the benchmark interface *callRate* operation on the remote machine.
5. The *BenchmarkControl* object records a transaction stop time.
6. The total time to complete the transaction is calculated by the *BenchmarkControl* object and saved to its measurement vector in memory.
7. The *BenchmarkControl* object repeats steps four through six 100 times to obtain measurement confidence before saving all test results as discussed previously (§5.4 Experiments).
8. These steps are repeated for a benchmark server process loaded on the same physical machine to determine the LPC dispatch rate.

5.4.1.2 Marshaling Rate

The objective of this experiment is to determine the rate that the MICO ORB can transmit and receive TFMS data types over the distributed network (§4.2.2.2 Marshaling Rate). Since the data-marshaling rate depends upon the data type being transmitted over the network, three marshaling rate transactions are executed [21]. The first transaction is for a double-precision floating-point number. The second transaction is for a TAF data structure. The last transaction concerns bulk TAF data structure transfers using the OMG IDL *sequence* type. All marshaling rate transactions implement steps one through seven of the call latency transaction, with the appropriate operation invoked in step four.

5.4.1.3 Server-Side Performance

In this experiment, the CORBA Event Loop (main thread) throughput is measured and compared to a thread-per-request multithreaded implementation (§4.2.2.3 Server-Side Performance). The experiment's objective is to determine the server workload where a multithreaded ORB implementation improves overall server throughput and response time for multiple clients.

5.4.1.3.1 Threaded

The threaded transaction uses the TFMS delegation layer (§3.5.2.4 Delegation Module) and is implemented using the following steps:

1. Prerequisites: Two machines are chosen for this test. The machine's clocks are synchronized with the Windows 2000 DC. A benchmark server and test generator process is loaded on different physical machines. The TFMS name server provides runtime location transparency.
2. As shown in Figure 42, the test generator instantiates a *BenchmarkControl* object as in previous tests. The *BenchmarkControl* object then instantiates a *ClientControl* object with a handle to the benchmark server process.
3. The *BenchmarkControl* records a transaction start time. Note: The *ClientControl* object inherits thread functionality from the *ThreadPool* object, which encapsulates all TFMS Win32 thread functionality. Figure 43 shows the interface for the *ThreadPool* object.

```
#include <windows.h>

class ThreadPool
{
public:
        ThreadPool();
        ~ThreadPool();
        static DWORD WINAPI Threaded(LPVOID);
        void CreateRunningThreads(int number);
        void CreateSuspendedThreads(int number);
        void StartThread();// starts suspended thread from pool
        void StartThreads();// starts entire thread pool
        void WaitForThreadsExit();// normally called after CreateSuspendedThreads
                                  // to control thread execution.

protected:
        HANDLE *_Threads;
        DWORD  _ThreadId;
        int _number;
        virtual DWORD ThreadFunction();// subclasses must override
        void AddThreadToPool();// maintains pool size
};
```

Figure 43: Thread Pool Class

4. The *BenchmarkControl* object calls the *ThreadPool CreateRunningThreads* member function, which creates a pool of 1, 2, 3, 4, and 5 threads (clients).

5. *Each* thread executes the overridden *ClientControl ThreadFunction* member function which calls the benchmark interface *multithread* operation and passes a 1, 10, 100, 1000, and 10000 (thousands of FLOPS) workload parameter to the server.
6. The benchmark server dispatches a *MultiThreadCommand* object to perform the client's requested computation. The *MultiThreadCommand* object also inherits thread functionality from *ThreadPool,* but calls *CreateRunningThreads(1)* in its constructor to start *one* thread immediately to perform the FLOP computation. A call to *WaitForThreadsExit* in its destructor ensures clean resource deallocation upon termination.
7. The *BenchmarkControl* object waits for all client threads to complete by calling the *ThreadPool WaitForThreadsExit* member function.
8. The total time to complete the transaction is calculated by the *BenchmarkControl* object and saved to its measurement vector in memory.
9. The *BenchmarkControl* object repeats steps four through seven 100 times to obtain measurement confidence before saving all test results (§5.4 Experiments).

5.4.1.3.2 Event Loop

The Event Loop transaction is implemented in exactly the same manner using a single-threaded server implementation. A *SingleThreadCommand* object is dispatched for each request, but this class uses the ORB's main thread (Event Loop) to execute a requested client's computation. The *SingleThreadCommand* does not execute in its own thread of control as the *MultiThreadCommand* object does in the multi-threaded experiment.

5.4.1.4 Name Server Performance

Name Server performance tests the effect of name binding and resolution as the number of name context bindings increases from one to 32,000.

5.4.2 Publication Transaction

The publication experiment considers replication algorithms suitable to perform the TFMS forecast replication function. Characteristics: 1) forecasts are validated at the originating OWS, so validated forecasts are treated as read-only data and 2) forecasts are replicated to *all* OWS and equivalent centers to meet performance and availability requirements for TFMS customers. Two replication algorithms are tested: 1) CORBA Event Service and 2) primary copy.

In the OMG Event Service, suppliers produce events and consumers receive events. Both suppliers and consumers connect to an event channel. The event channel is *central* to the event service and conveys events between suppliers and consumers without requiring suppliers and consumers to have knowledge of one another [21]. Four models support event delivery: canonical push, canonical pull, and two hybrid push/pull or pull/push models.

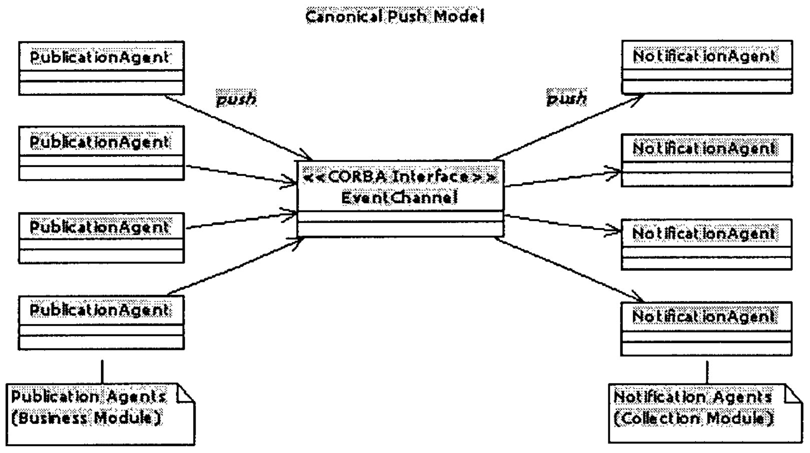


Figure 44: Canonical Push Model

Figure 44 is a representation of the canonical push model that maps to the distributed characteristics as listed above. In the canonical push model, publication agents (suppliers) push events to the event channel. The event channel pushes these events in turn to notification agents (consumers).

In the primary copy algorithm shown in Figure 45, a particular publication agent serves as the *primary* server for forecasts entered and validated in their region. The publication agent serves as the coordinator and must have *knowledge* of notification agents or collection modules (depending on implementation) to replicate forecasts to them. The publication agent is multi-threaded to improve performance.

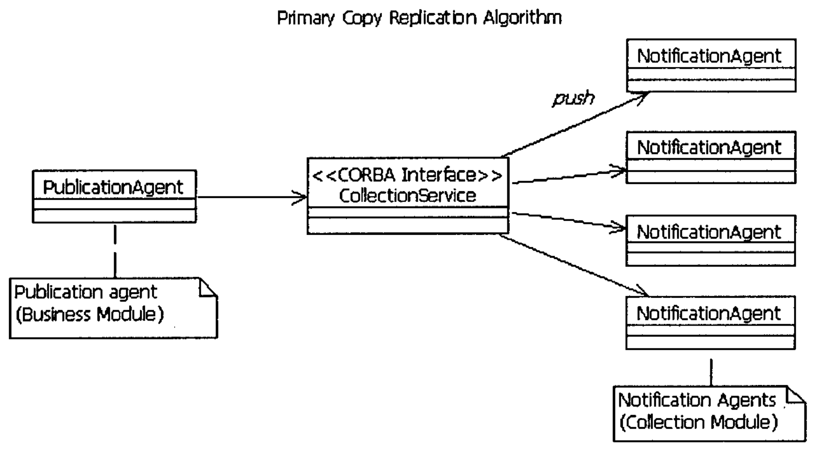


Figure 45: Primary Copy Algorithm

An alternative algorithm is shown in Figure 46, where the replication is performed in tree-like fashion. Each publication agent sends forecasts to only a few other OWS centers (its children), who then send forecasts to its children and so forth. This forms a hierarchical pipeline distribution pattern where knowledge of other notification agents is kept to a minimum.

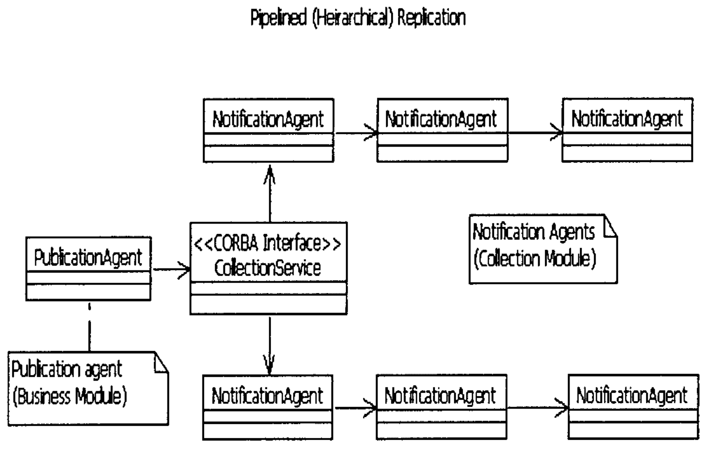
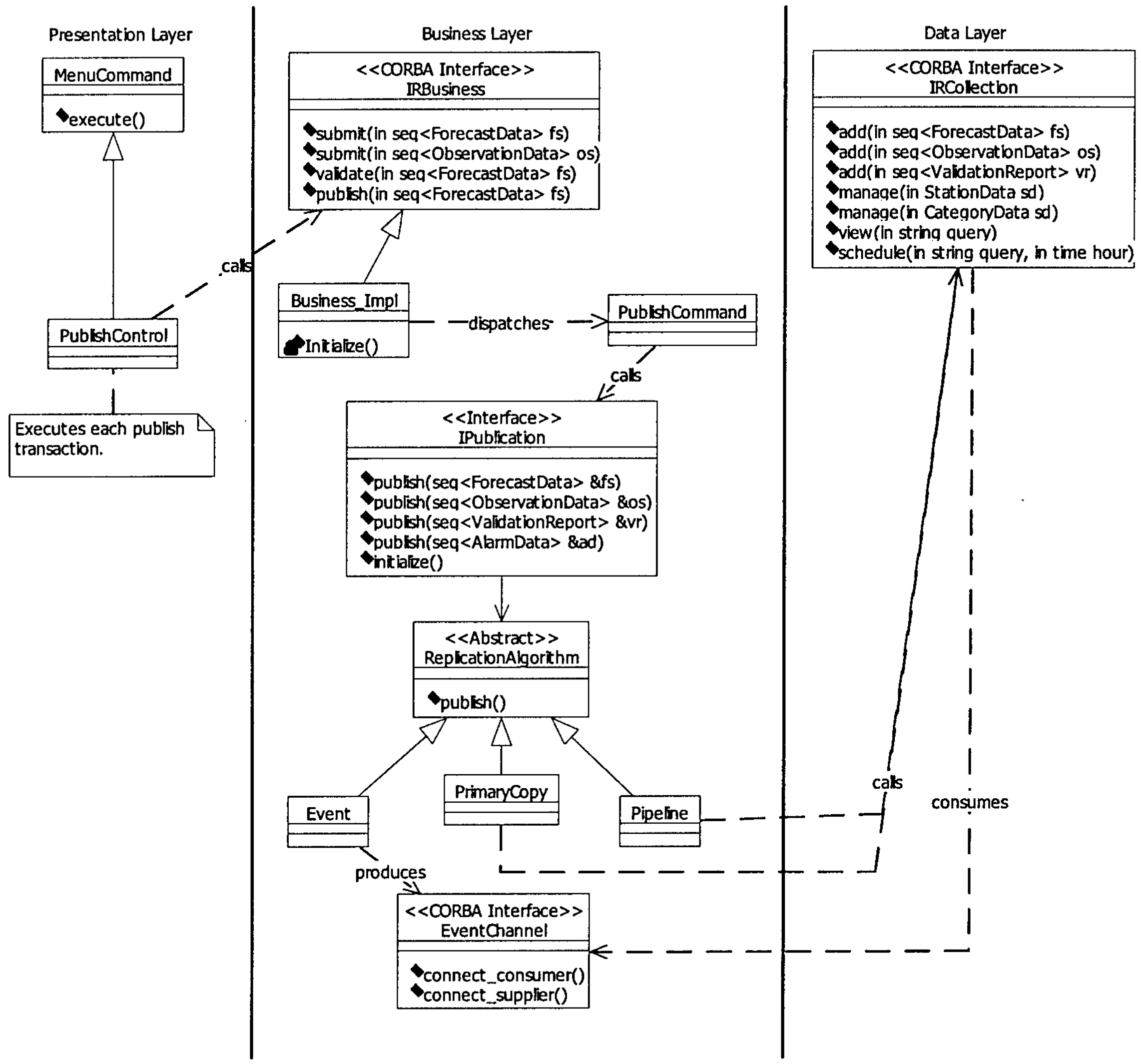


Figure 46: Pipelined Publication

The objective of this experiment is to determine which replication algorithm is suitable for TFMS forecast data based on *minimum communication time* and *maximum flexibility*. Publication transactions involve these steps:

1. Prerequisites: Up to eight machines are chosen for this test. The machine's clocks are synchronized with the Windows 2000 DC. The test generator and one business server are loaded on one machine; and 3, 5, and 7 collection servers are loaded on different physical machines. The collection and business servers all bind to the TFMS name server on startup.
2. The test generator instantiates a *PublishControl* object.
3. The *PublishControl* object records a transaction start time, and then calls the business server's *publish* method sending a sequence of 1, 10, 100, and 1000 forecasts as shown in Figure 47.
4. The business server instantiates a *PublishCommand* object, passing it the forecast data and algorithm to perform.
5. The *PublishCommand* object calls the publication module's *publish* method, passing along the forecast data and algorithm.
6. If the algorithm is primary copy, the collection server's *add* method is called with the sequence of forecast data. The collection server instantiates a *Forecast* object for each forecast then inserts the object in the cache collection to complete the transaction.



Presentation Layer
Business Layer
Data Layer
MenuCommand
execute()
PublishControl
Executes each publish transaction.
calls
<<CORBA Interface>>
IRBusiness
submit(in seq<ForecastData> fs)
submit(in seq<ObservationData> os)
validate(in seq<ForecastData> fs)
publish(in seq<ForecastData> fs)
Business_Impl
Initialize()
dispatches
PublishCommand
calls
<<Interface>>
IPublication
publish(seq<ForecastData> &fs)
publish(seq<ObservationData> &os)
publish(seq<ValidationReport> &vr)
publish(seq<AlarmData> &ad)
initialize()
<<Abstract>>
ReplicationAlgorithm
publish()
Event
PrimaryCopy
Pipeline
produces
<<CORBA Interface>>
EventChannel
connect_consumer()
connect_supplier()
<<CORBA Interface>>
IRCollection
add(in seq<ForecastData> fs)
add(in seq<ObservationData> os)
add(in seq<ValidationReport> vr)
manage(in StationData sd)
manage(in CategoryData sd)
view(in string query)
schedule(in string query, in time hour)
calls
consumes

Figure 47: Publication Transactions

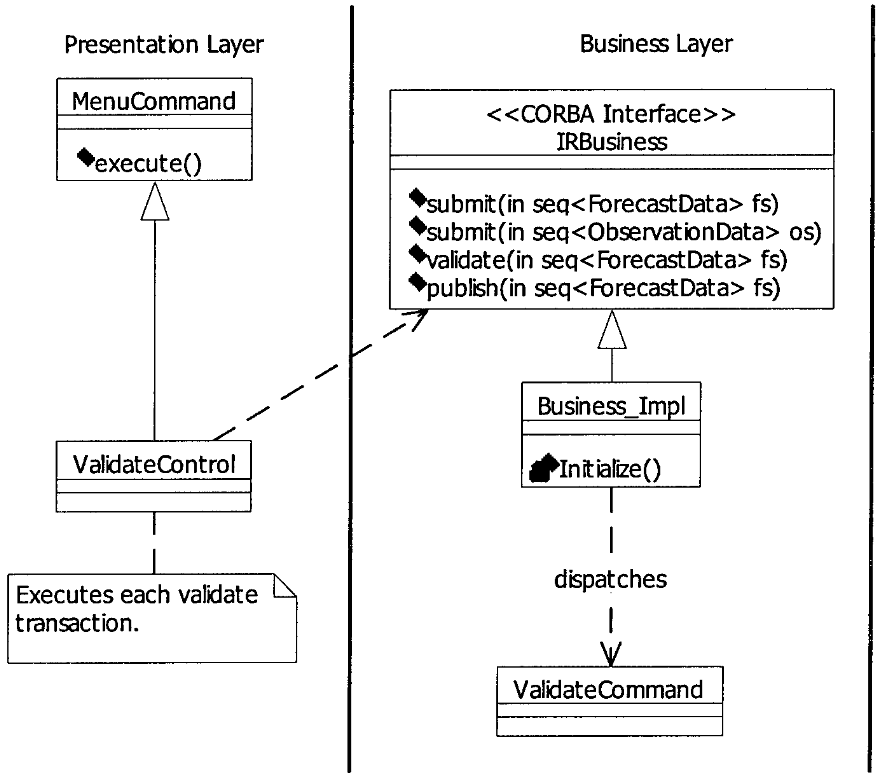
7. If the algorithm is event channel, the sequence of forecast data is published using the CORBA Event Channel. A *NotificationAgent* receives the sequence of forecast data using the Event Channel, then calls the add method for insertion as in the previous step.

8. The total time to complete the transaction is calculated by the *PublishControl* object and written to a file upon test completion. The *PublishControl* object

repeats this sequence 100 times to obtain measurement confidence before returning control back to the experimenter.

5.4.3 Validation Transaction

Figure 48 shows the validation transaction. The objective of this experiment is to determine the execution time and efficiency for forecast validation using single forecasts, then bulk validation requests of 10, 100, and 1000 forecasts. The forecast validation function is then distributed over 2, 3, 4, and 5 processors to determine if the TFMS validation function will benefit from distributed processing methods.



Presentation Layer
Business Layer
MenuCommand
execute()
<<CORBA Interface>>
IRBusiness
submit(in seq<ForecastData> fs)
submit(in seq<ObservationData> os)
validate(in seq<ForecastData> fs)
publish(in seq<ForecastData> fs)
Business_Impl
Initialize()
ValidateControl
dispatches
Executes each validate transaction.
ValidateCommand

Figure 48: Validation Transaction

The validation transaction is implemented using the following steps:

1. Prerequisites: Six machines are chosen for this test. The machine's clocks are synchronized with the Windows 2000 DC. One test generator and five business servers are loaded on different physical machines. The business servers all bind to the TFMS name server on startup.
2. The experimenter selects the validation test when prompted by the test generator.
3. As shown in Figure 48, the test generator instantiates a *ValidateControl* object. The test generator creates and synchronizes 1, 2, 3, 4, or 5 threaded *ValidateControl* objects as validation control agents. Each *ValidateControl* object is instantiated with a handle to a business server process.
4. Once all threads are created and synchronized, the test generator records a transaction start time and then releases the thread barrier. Each *ValidateControl* object calls the *IRBusiness* interface *validate* operation and passes the server 1, 10, 100, or 1000 forecasts.
5. The total time to complete all validation requests is calculated by the test generator and written to a file upon test completion. The test generator repeats this sequence 100 times to obtain measurement confidence before returning control back to the experimenter.

5.4.4 Submit Transaction

Integration testing involves testing TFMS module interfaces to make sure they actually can communicate with each other without losing data or sacrificing system functionality [6]. System testing involves functional, load, and performance tests of the integrated TFMS distributed software system to ensure *customer* requirements have been *effectively* met [76]. In this research, one effectiveness transaction is implemented: the submit transaction (§3.4.2.4 Transaction Model).

Figure 49 shows the submit transaction. The objective of this experiment is to determine the execution time for submitting forecasts for validation, publication, and data collection within an OWS regional processing center. As with the validation experiment, single forecasts, then bulk update operations of 10, 100, and 1000 forecasts are tested. The validation function is distributed over 1, 3, and 5 processors to determine if distributed processing is appropriate within an OWS site.

The submit transaction is implemented using the following steps:

1. Prerequisites: Seven machines are chosen for this test. The machine's clocks are synchronized with the Windows 2000 DC. One test generator, one collection server, and five business servers are loaded on different physical machines. The collection and business servers all bind to the TFMS name server on startup.
2. The experimenter selects the integration test when prompted by the test generator.
3. As shown in Figure 49, the test generator instantiates a *SubmitControl* object. The *SubmitControl* object creates and synchronizes 1, 3, or 5 threaded *Submit-*

Thread objects as update control agents. Each *SubmitThread* object is provided a handle to a business server process.

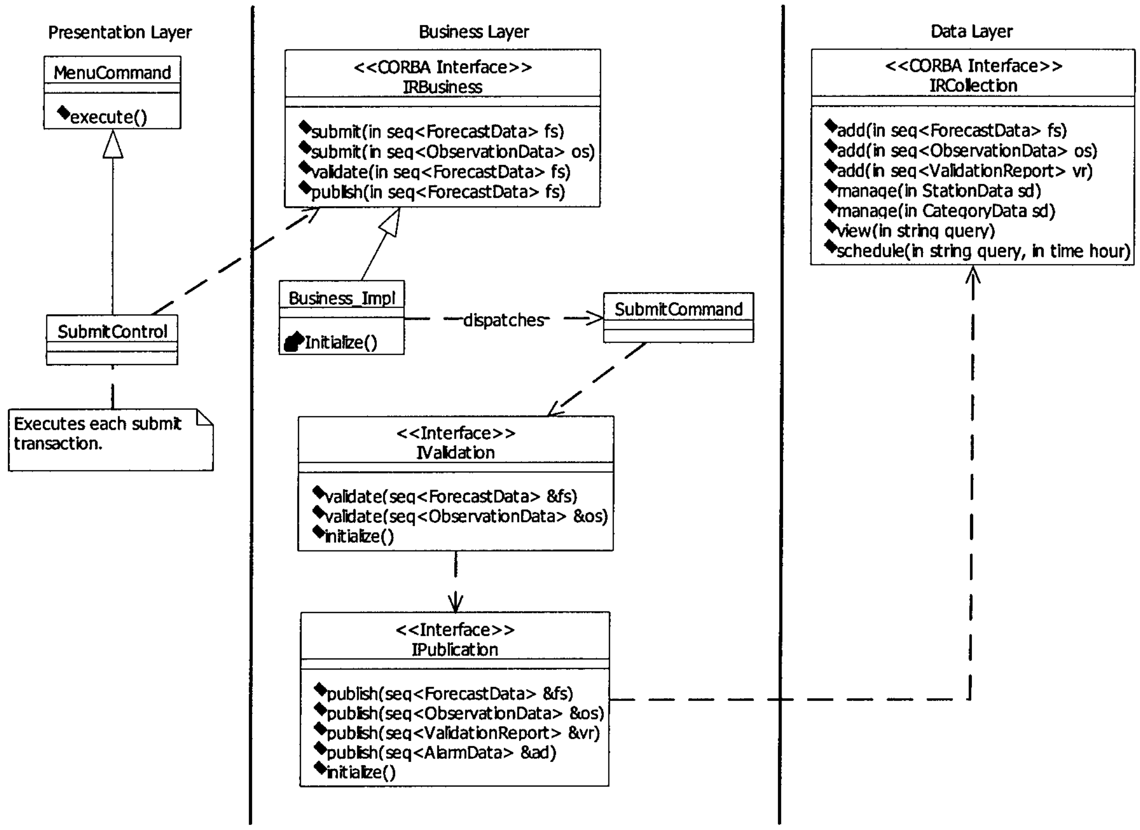

Presentation Layer
Business Layer
Data Layer
MenuCommand
execute()
SubmitControl
Executes each submit transaction.
<<CORBA Interface>> IRBusiness
submit(in seq<ForecastData> fs)
submit(in seq<ObservationData> os)
validate(in seq<ForecastData> fs)
publish(in seq<ForecastData> fs)
Business_Impl
Initialize()
dispatches
SubmitCommand
<<Interface>> IValidation
validate(seq<ForecastData> &fs)
validate(seq<ObservationData> &os)
initialize()
<<Interface>> IPublication
publish(seq<ForecastData> &fs)
publish(seq<ObservationData> &os)
publish(seq<ValidationReport> &vr)
publish(seq<AlarmData> &ad)
initialize()
<<CORBA Interface>> IRCollection
add(in seq<ForecastData> fs)
add(in seq<ObservationData> os)
add(in seq<ValidationReport> vr)
manage(in StationData sd)
manage(in CategoryData sd)
view(in string query)
schedule(in string query, in time hour)

Figure 49: Submit Transaction

4. Each *SubmitControl* object calls the *IRBusiness* interface *submit* operation and passes the server 1, 10, 100, or 1000 forecasts for entry into the TFMS distributed system.
5. Once validated, the validation module publishes the forecasts using the Event Channel as previously discussed.
6. The total time to complete all update operations is calculated by the test generator and written to a file upon test completion. The test generator repeats

this sequence 100 times to obtain measurement confidence before returning control back to the experimenter.

5.5 Summary

In this chapter, the distributed TFMS implementation details are discussed in great detail. Both experimental operating environment and transaction characteristics are presented and documented. The N-tier topology used to test the TFMS is also shown and implementation issues concerning various distributed topologies along with their inherent component placement tradeoffs are also explained.

VI. Collected Data Analysis

The purpose of computing is insight, not numbers. - R.W. Hamming

6.1 Introduction

In this chapter, results based on experimental design and implementation are presented and discussed (§4.3 Experimental Design, §5.4 Experiments). CORBA benchmark transaction results are shown and the impact of measuring call dispatch rate, marshaling rate, server-side throughput, and Name Server performance is discussed in the context of overall distributed system design. Forecast publication and validation results are presented and the efficacy of flexible system design is addressed in the context of federating these services within the distributed TFMS [21].

6.2 ORB Experiments

The objective of the call latency experiment is to determine the cost of a MICO ORB remote call relative to a local call in the same address space. These measurements enable relative efficiency to be computed to aid component placement decisions within a *particular* operating environment (§5.3.2 Experimental Environment). The dispatch rate that a particular ORB can deliver sets a fundamental design limit on a distributed application [21].

Figures 50 and 51 show a remote call latency of 0.000088 seconds and a local call latency of 0.00000081 seconds, which correspond to *maximum* call dispatch rates of 11,364 calls/sec for the MICO ORB and 1,234,568 calls/sec for local calls. These results reflect the fact that a remote call using the MICO ORB is 100 times more expensive than a local call.

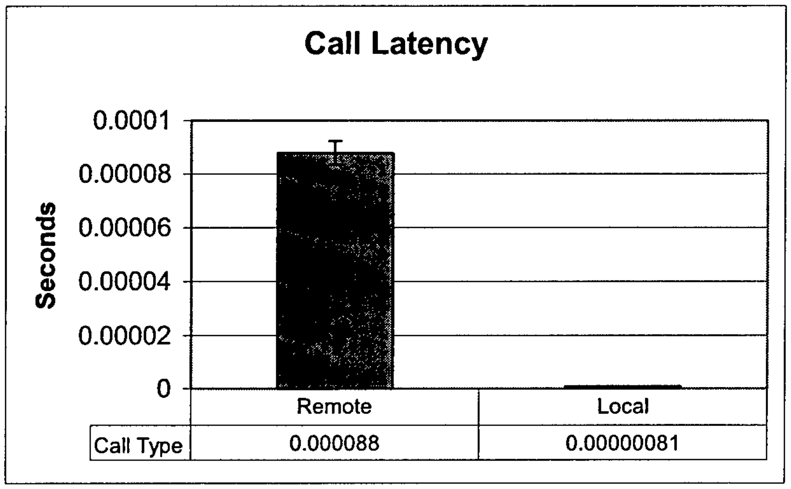


Figure 50: Remote and Local Procedure Call Latency

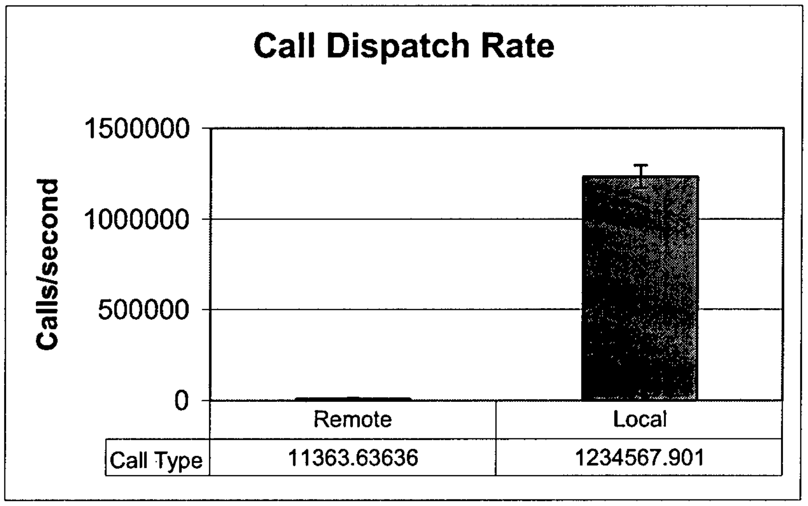
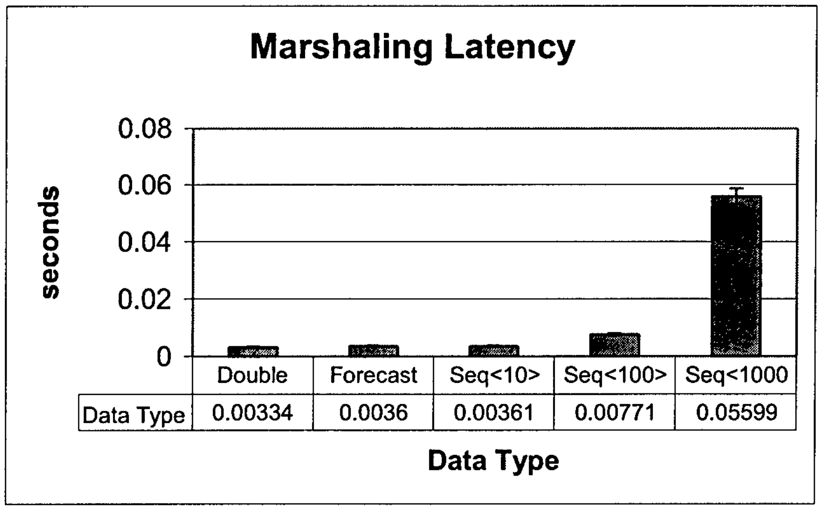


Figure 51: Remote and Local Procedure Call Rates

The Relative Efficiency (RE) for a *parameterless* remote operation in this experimental environment is given by (§4.2.2.1 Call Latency):

RE~= Operation runtime/(0.000088 + Operation runtime)

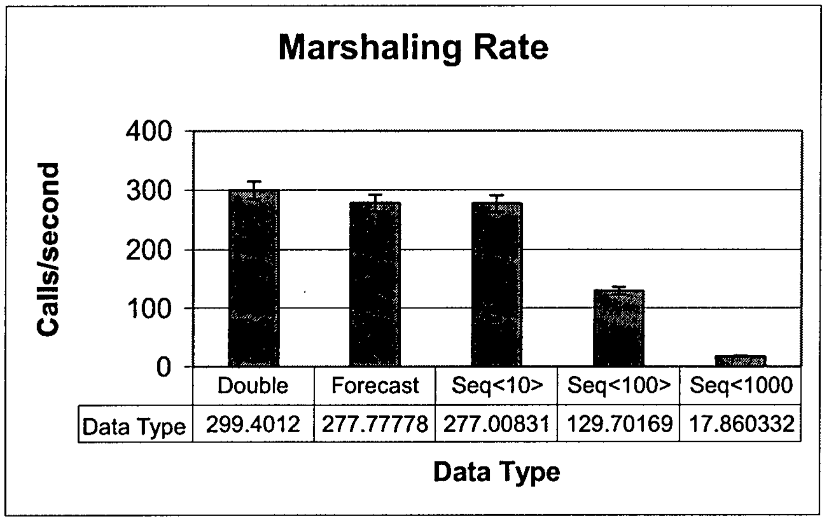
The objective of the marshaling rate experiments is to determine the cost of transmitting and receiving various data types used within the TFMS experimental environment (§5.3.2 Experimental Environment). These measurements enable a designer to determine the most efficient way to pass data within a given distributed environment. The dispatch rates for various data types communicated in the distributed system also serve as fundamental design limits for a particular ORB.



	Double	Forecast	Seq<10>	Seq<100>	Seq<1000
Data Type	0.00334	0.0036	0.00361	0.00771	0.05599

Figure 52: MICO ORB Marshaling Latency

Figures 52 and 53 shows the latency and dispatch rates associated with marshaling double, single Forecast, 10 Forecast sequence, 100 Forecast sequence, and 1000 Forecast sequence data types. Latency and dispatch rates are fairly consistent for the double, single Forecast, and 10 Forecast sequence data types, but latency noticeably increases and the corresponding dispatch rates noticeably decrease for larger data types.

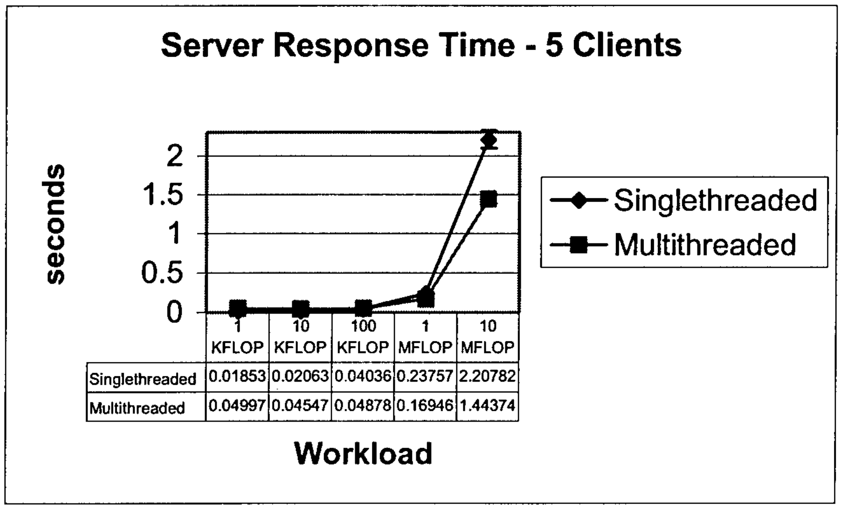


	Double	Forecast	Seq<10>	Seq<100>	Seq<1000
Data Type	299.4012	277.77778	277.00831	129.70169	17.860332

Figure 53: MICO ORB Marshaling Rates

These results follow from using an Ethernet network topology (§4.3.1 Factors, §5.3.2 Experimental Environment). An Ethernet frame can transmit a maximum of 1500 bytes of data [10]. Using the C++ *sizeof* operator, a TFMS Forecast data type is 108 bytes and a MICO sequence data type is 16 bytes. A 10 Forecast sequence requires 1096 data bytes, so this easily fits within one Ethernet frame and does not cause increased latency due to data fragmentation as indicated by the 100 or 1000 Forecast sequence results. To find the sequence size or multiple of this size, which maximizes data transmission efficiency, knowledge of the network topology and data characteristics, is required. Since the Ethernet data frame consists of a 1500 byte maximum, a MICO sequence type is 16 bytes, and a TFMS Forecast is 108 bytes, (1500 - 16)/108 or a 13 Forecast sequence makes the most use of the given network topology.

The objective of comparing a single-threaded with a thread-per-request server implementation is to determine the point where multithreading an application results in improved client response time and throughput (§4.2.2.3 Server-Side Performance). These measurements enable a designer to determine if a multithreaded approach improves performance for a particular operation or module within a given distributed environment.



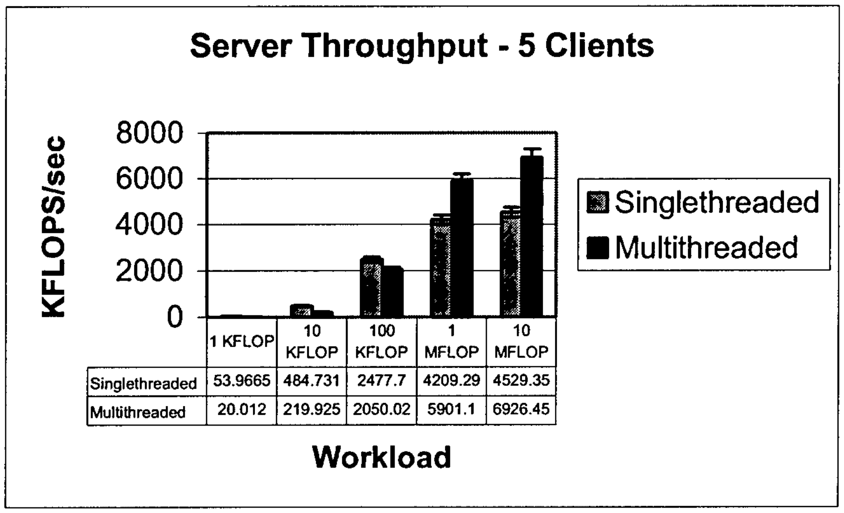
	1 KFLOP	10 KFLOP	100 KFLOP	1 MFLOP	10 MFLOP
Singlethreaded	0.01853	0.02063	0.04036	0.23757	2.20782
Multithreaded	0.04997	0.04547	0.04878	0.16946	1.44374

Figure 54: Server Response Time Comparison – 5 Clients

Figures 54 and 55 shows the response time and throughput for 1, 10, 100, 1000, and 10000 KFLOP workloads executed on a dual PIII 550Mhz machine with 256MB of memory. The workload that fully utilizes the server is determined by computing the efficiency of the operation. Using the remote call latency measurement of 0.000088 seconds to compute relative efficiency, Table 8 shows that a single-threaded server is underutilized for the 10 KFLOP workload, but is fully utilized for the 1 MFLOP workload!

Table 8: Workload Efficiencies

Workload	Execution Time	Efficiency
10 KFLOP	0.00047	0.84
100 KFLOP	0.00455	0.98
1 MFLOP	0.0454	0.998

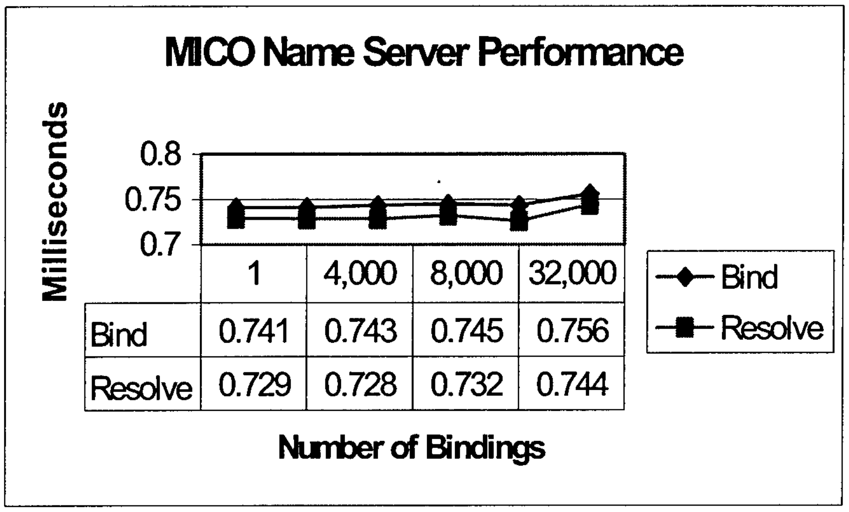


	1 KFLOP	10 KFLOP	100 KFLOP	1 MFLOP	10 MFLOP
Singlethreaded	53.9665	484.731	2477.7	4209.29	4529.35
Multithreaded	20.012	219.925	2050.02	5901.1	6926.45

Figure 55: Server Throughput Comparison – 5 Clients

Figures 54 and 55 support this analysis where it's clear that multithreading the server decreases response time and increases overall throughput as workloads increase above 100 KFLOP. These results follow from using a symmetrical processor, i.e. a higher degree of machine scalability, where the multi-threaded application scales better with the additional processor. This improvement is due to true concurrency, where threads are executing at the same time on different processors.

The purpose of OMG Naming Service tests is to determine if name binding and resolution latency is dependent on the number of bindings in a single name context as the name bindings increases from one to 32,000. This test helps designers determine the impact and cost of binding and resolving CORBA object references to a given vendor's Naming Service implementation.



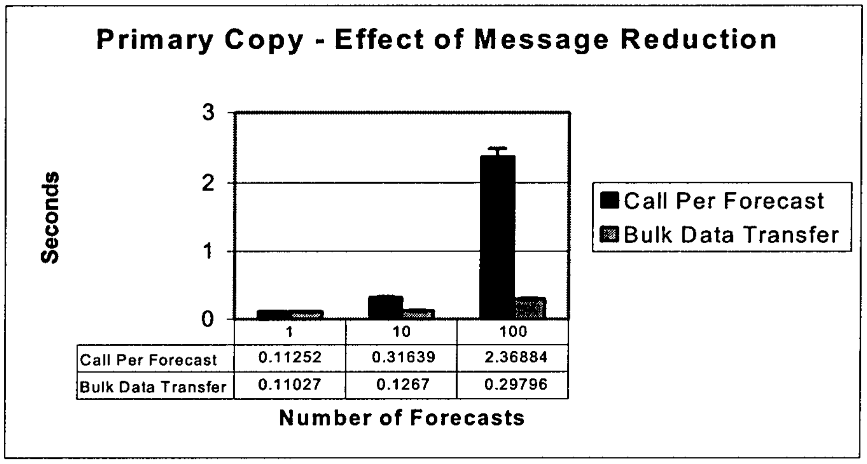
	1	4,000	8,000	32,000
Bind	0.741	0.743	0.745	0.756
Resolve	0.729	0.728	0.732	0.744

Figure 56: MICO Name Server Performance

As Figure 56 illustrates, the MICO naming service implementation shows no dramatic increase in name binding or resolution latency as the number of name bindings increases. This indicates that binding and resolution latency is not dependent on the number of name bindings. Measurement variation for the experiment is 0.1 percent.

6.3 Publication Experiments

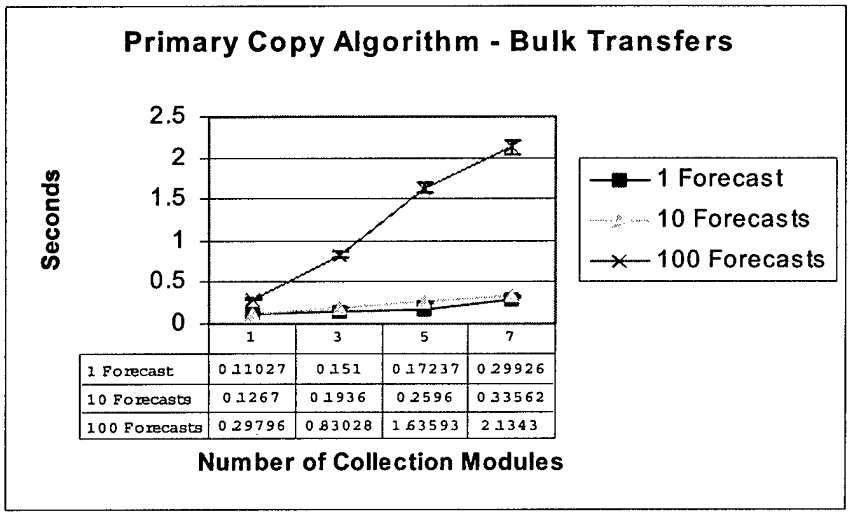
The objective of TFMS publication experiments is to determine a suitable replication algorithm to disseminate TFMS weather data, based on *minimum communication time* and *maximum flexibility*. First, the effect of call overhead is investigated and its impact on TFMS distributed system design is shown. Figure 57 shows the cost of sending 1, 10, and 100 singular forecasts as compared to a bulk transfer of equal size. The publication results for a single forecast start off equal, but larger forecast transfers result in excessive communication costs associated with the marshaling latency for a single forecast (§6.2 ORB Experiments). These results confirm the importance of designing remote interface operations with bulk data transfer support.



	1	10	100
Call Per Forecast	0.11252	0.31639	2.36884
Bulk Data Transfer	0.11027	0.1267	0.29796

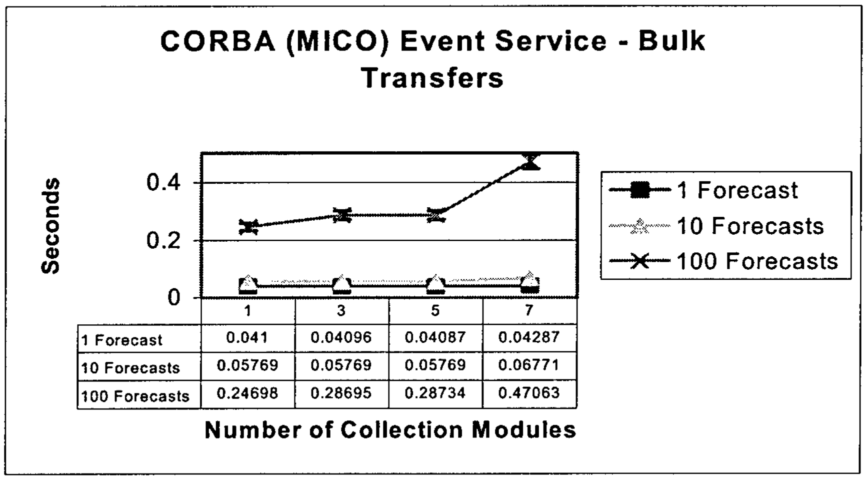
Figure 57: Effect of Message Reduction

The primary copy and the MICO OMG Event Service publication execution times for 1, 10, and 100 forecasts are shown in Figures 58 and 59. The number of collection servers (receiving sites) ranged from one to seven. The primary copy and Event Service algorithm used *one* business server to publish Forecasts to collection servers.



	1	3	5	7
1 Forecast	0.11027	0.151	0.17237	0.29926
10 Forecasts	0.1267	0.1936	0.2596	0.33562
100 Forecasts	0.29796	0.83028	1.63593	2.1343

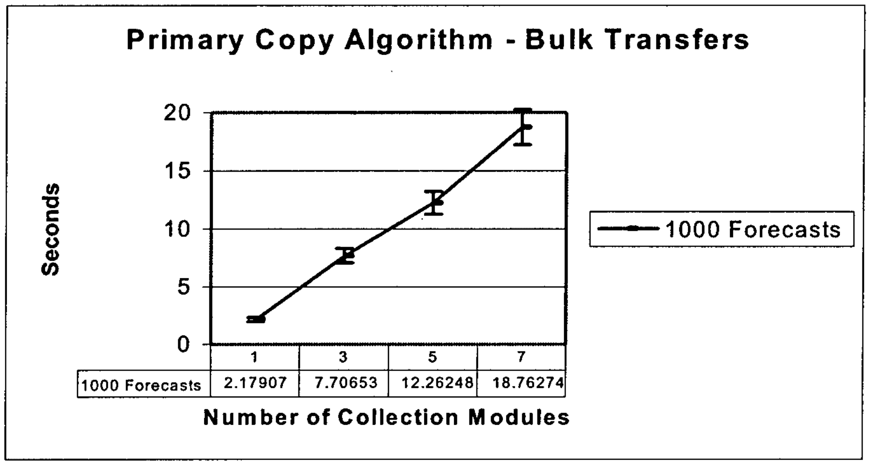
Figure 58: Primary Copy Bulk Data Transfers – 1, 10, 100 Forecasts



	1	3	5	7
1 Forecast	0.041	0.04096	0.04087	0.04287
10 Forecasts	0.05769	0.05769	0.05769	0.06771
100 Forecasts	0.24698	0.28695	0.28734	0.47063

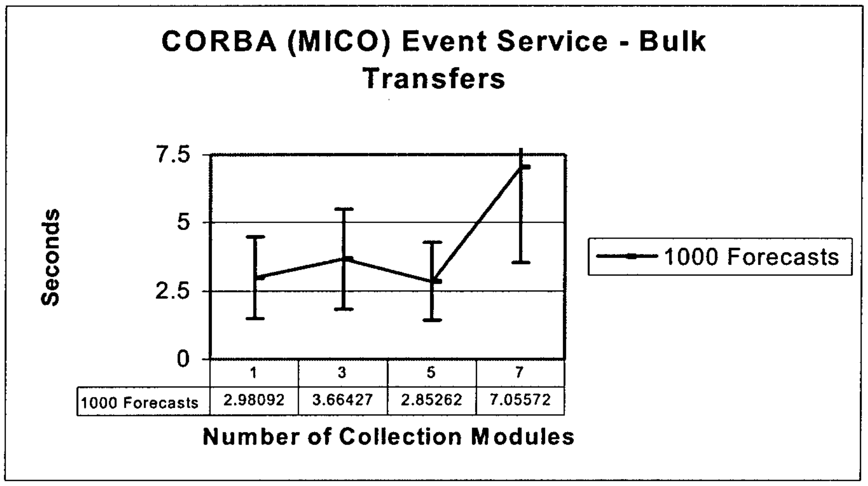
Figure 59: MICO Event Service Bulk Data Transfers – 1, 10, 100 Forecasts

As shown in Figure 58 and 60, the primary copy execution times increased linearly with the number of forecasts and collection servers. The MICO Event Service, shown in Figures 59 and 61, displays the lowest communication time and exhibits problem-size scalability when publishing all forecast data sizes to 1, 3, or 5 collection sites. However, publishing 100 or 1000 forecasts to seven collection sites does place a noticeable load on the publication event channel. If using an Event Service, it's important to test its fan-out capacity, which typically means to further stress the channel by increasing event occurrence or size (forecasts) and consumer size (collection sites). The MICO Event Service is scalable for all forecast sizes up to five collection sites, but larger data sizes of 100 and 1000 forecasts stress the event channel when additional consumers subscribe for publication events. Figures 59 and 61 show this behavior for these larger data and consumer sizes.



	1	3	5	7
1000 Forecasts	2.17907	7.70653	12.26248	18.76274

Figure 60: Primary Copy Bulk Data Transfers – 1000 Forecasts



	1	3	5	7
1000 Forecasts	2.98092	3.66427	2.85262	7.05572

Figure 61: MICO Event Service Bulk Data Transfers – 1000 Forecasts

As stated previously (§5.4.2 Publication Transaction), the primary copy algorithm requires knowledge of collection server names in order to obtain module references from the TFMS Name server. While the use of a name server does provide location independence by resolving references at runtime, the Event Service has the advantage of completely decoupling publishers from consumers, adding to business and collection module autonomy [12 and 21].

6.4 Validation Experiments

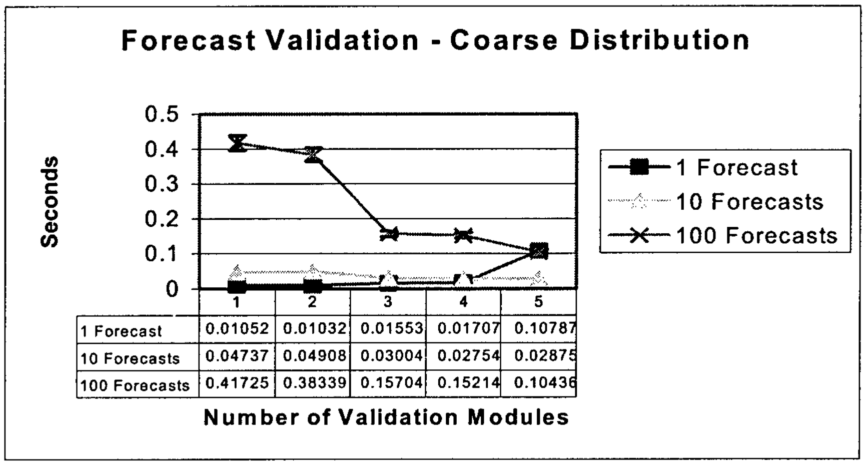
The objective of forecast validation experiments is to determine the execution time for forecast validation using single forecasts, then bulk validation requests of 10, 100, and 1000 forecasts on a single machine. The validation function is then distributed over 2, 3, 4, and 5 processors to determine the benefit of distributed processing, if any. As discussed previously (§3.4.2.5 Granularity), individual validation rules may be distributed, or the entire module itself. This experiment considered distribution of the validation module, which requires 0.0044 seconds to validate one forecast.

Figures 62 and 64 show the execution times for validating 1, 10, 100, and 1000 forecasts using 1, 2, 3, 4, or 5 validation modules. An important aspect of distributed processing is scalability, or the ability to maintain efficiency as the problem size and number of processors increase [15]. Figures 63 and 65 show the corresponding efficiency for Figures 62 and 64.

Efficiency is given by:

$$E = Ts/pTp$$

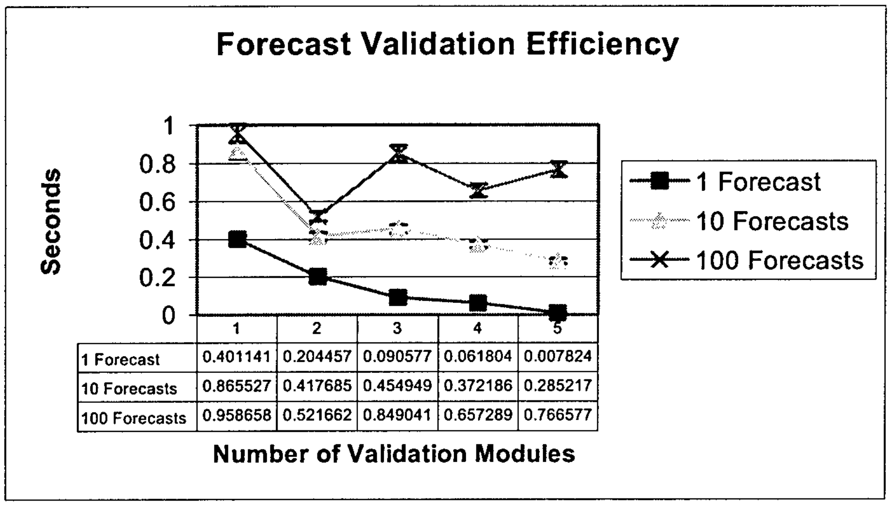
Where Ts is the serial execution time, p is the number of processors, and Tp is the parallel execution time [14]. Tp is further decomposed into a computational (Ts) and overhead (To) component. Marshaling latency is *part* of the overhead component in this equation.



	1	2	3	4	5
1 Forecast	0.01052	0.01032	0.01553	0.01707	0.10787
10 Forecasts	0.04737	0.04908	0.03004	0.02754	0.02875
100 Forecasts	0.41725	0.38339	0.15704	0.15214	0.10436

Figure 62: Validation Distribution – 1, 10, 100 Forecasts

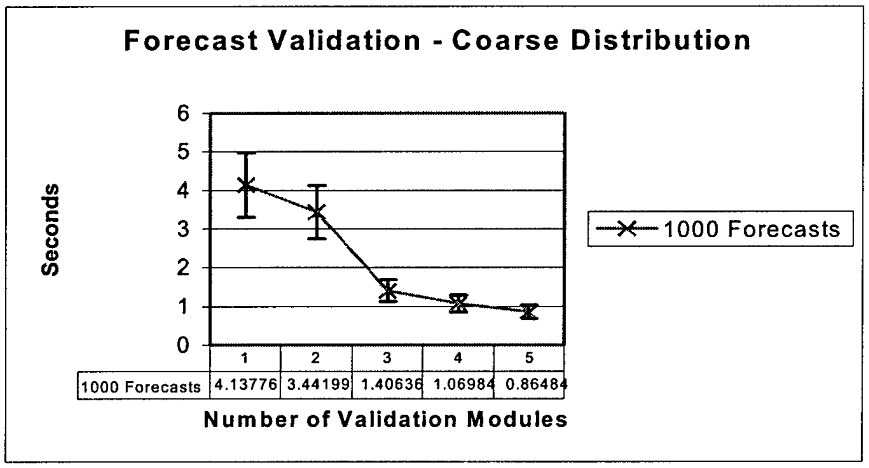
As shown in Figures 62 and 63, the 100 forecast problem size definitely benefits from distributed processing with its corresponding efficiency generally maintained for increases in the number of processors. One and ten forecast sizes do *not* maintain their corresponding efficiencies as the number of processors are increased; this indicates that overhead time is dominating useful computational (validation) time for these problem sizes.



	1	2	3	4	5
1 Forecast	0.401141	0.204457	0.090577	0.061804	0.007824
10 Forecasts	0.865527	0.417685	0.454949	0.372186	0.285217
100 Forecasts	0.958658	0.521662	0.849041	0.657289	0.766577

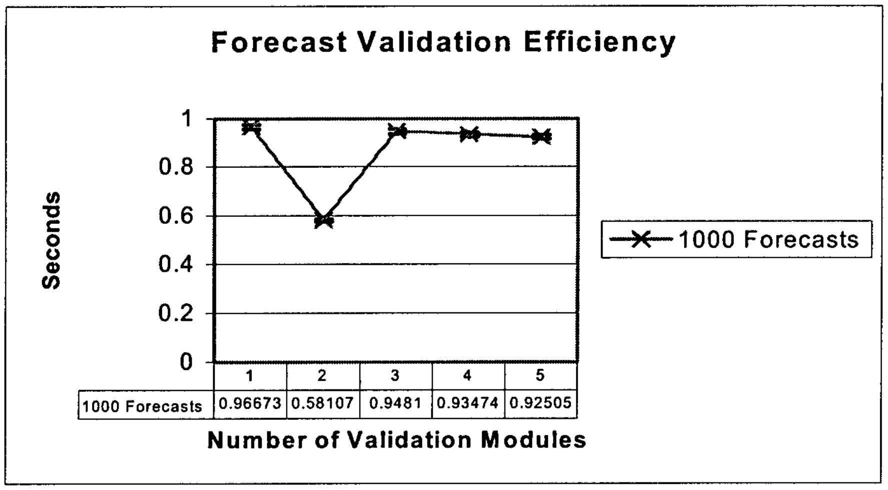
Figure 63: Validation Efficiency – 1, 10, 100 Forecasts

In Figures 64 and 65, the 1000 forecast problem size also shows a decrease in overall execution time with its corresponding efficiency generally maintained for increases in the number of processors.



	1	2	3	4	5
1000 Forecasts	4.13776	3.44199	1.40636	1.06984	0.86484

Figure 64: Validation Distribution – 1000 Forecasts



	1	2	3	4	5
1000 Forecasts	0.96673	0.58107	0.9481	0.93474	0.92505

Figure 65: Validation Efficiency – 1000 Forecasts

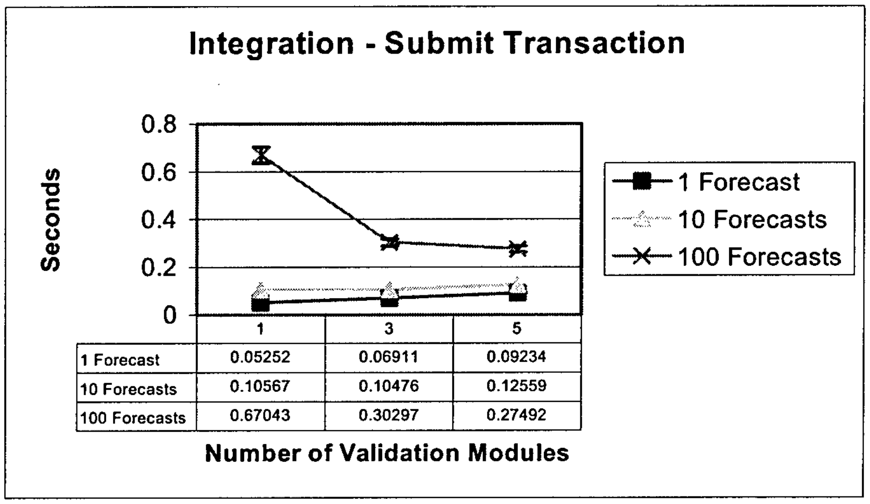
An interesting experiment result is the sharp drop in efficiency for all problem sizes when the number of processors is two. Once again, this is due to the operation's overhead time dominating computational time. For example, in the case of a 1000 forecast problem size for two processors, 500 forecasts are sent to each processor for validation. For three processors, 333 forecasts are sent to two processors, while 334 are sent to the third, and so on.

The validation experiment uses a multi-threaded client to distribute the statically balanced load to on-line validation modules, i.e. a separate thread is used to send the forecast load to each validation module and is subject to the scheduling policies of the underlying operating system, Win32 in this case. Each thread has its own overhead associated with it, e.g. creation time, waiting to run time, communication time. Since there is no distinction in thread priority, Win32 uses a time quantum (round-robin) scheduling scheme to give each thread execution time [11]. In the case of two processors the thread's overhead associated with buffering and transmitting larger forecast sizes, appear to cause the lower efficiency relative to using a higher number of processors with lesser communication requirements. The forecast size is also substantial enough in the two-processor case to cause a lower efficiency than the single processor, since both threads compete for processor time, while no competition occurs for the single processor. These results show that for larger forecast sizes, validation distribution is scalable when using 3, 4, or 5 processors, but efficiency suffers when using two processors for forecast validation, assuming a multi-threaded distribution mechanism as previously discussed.

6.5 Integration and System Experiments

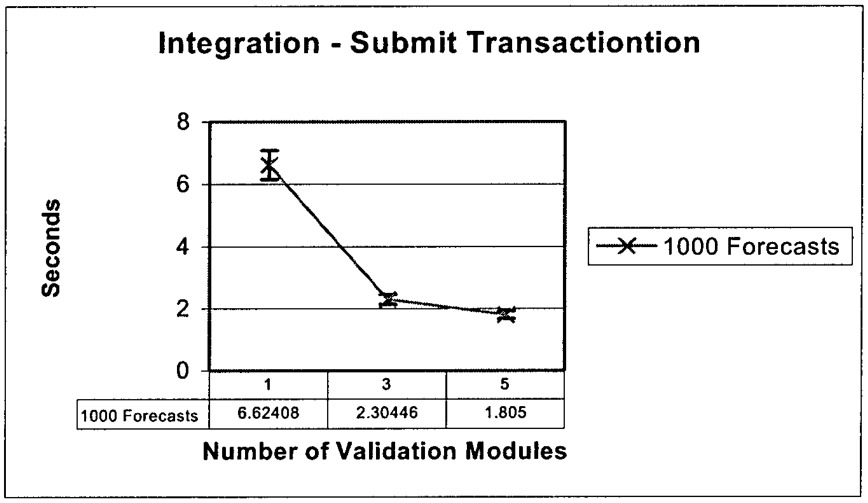
The objective of integration and system experiments is to put the TFMS modules together and test system identified transactions (§3.4.1.2 Functional Description). This research tests the submit transaction (§3.4.2.4 Transaction Model, §5.4.4 Submit Transaction). As with the validation experiment, single forecasts, then bulk update operations of 10, 100, and 1000 forecasts are tested. The submit transaction is deployed using a N-tier topology (§5.3.1 N-Tier Topologies), where 1, 3 and 5 processors are used as business layers, which validate and publish forecasts using the Event channel algorithm, and one processor is used as the data layer, which adds forecasts to an object cache.

As shown in Figures 66 and 67, execution times decrease significantly when submitting 100 and 1000 forecasts to three processors, but the rate of decrease starts to flatten out for five processors. These results generally follow the validation experiment results, since publication using the MICO Event channel is scalable for all forecast sizes up to five collection sites.



	1	3	5
1 Forecast	0.05252	0.06911	0.09234
10 Forecasts	0.10567	0.10476	0.12559
100 Forecasts	0.67043	0.30297	0.27492

Figure 66: Submit Transaction – 1, 10, 100 Forecasts



	1	3	5
1000 Forecasts	6.62408	2.30446	1.805

Figure 67: Submit Transaction – 1000 Forecasts

For one and ten forecasts, distributed processing has *no* benefit at all, as execution time gradually increases with additional business (validation) layers.

6.6 Summary

In this chapter, results based on experimental design and implementation are presented and discussed (§4.3 Experimental Design, §5.4 Experiments). CORBA benchmark results are shown and the impact of measuring call dispatch rate, marshaling rate, server-side throughput, and Name Server performance is discussed in the context of overall distributed system design. Forecast publication and validation results are presented and the efficacy of flexible system design is addressed in the context of federating these services within the distributed TFMS [21]. Integration experiments are used to show the effectiveness in meeting the customer's functional requirements, and showing why a certain topology or processor allocation is employed.

VII. Conclusion

A complex system that works is invariably found to have evolved from a simple system that worked. -- John Gall

7.1 Research Significance Revisited

This research provides a distributed system engineering approach for the Air Force Weather Agency (AFWA), which uses a combination of structured, object-oriented, and distributed software engineering techniques to develop an efficient and evolutionary software system [4, 5, 6, 16, and 17]. This design methodology incorporates proven principles such as component reuse and architectural development through the use of design patterns and modular software construction techniques [13 and 14]. While these design techniques are nothing new to most software developers, the methodology provides AFWA with a *measured* approach that examines the *big picture* of integrating their business functions into a truly *global* software environment. This investigation addresses TAF submission, validation, publication, *and* metric collection functions *in their entirety,* showing exactly how these functions are decomposed, designed, implemented, and ultimately allocated to hardware within a distributed computing environment. By viewing the problem in a global context, research goals and objectives were surpassed and the results are a more realistic representation of possible solutions (§1.3 Research Goals, §1.4 Specific Objectives), given the constraint of an open, distributed AFW operating environment (§1.6 Assumptions, Scope, and Constraints).

In addition to providing an *end-to-end* investigation of TAF processing and metric collection, this effort also provides AFW with a logical, rule-based measurement model used to measure and collect data quality statistics. This measurement model has the potential impact of providing each regional processing center (OWS) with its own data quality collection facility to measure the data processing process while also certifying data prior to submission to the data warehouse at AFWA (§3.4.1.5 Measuring TAF Data Quality and Accuracy). The effectiveness of this data measurement and processing model is one of using a simple, consistent approach to processing and measuring data at the *local* (regional) level. The regional processing centers must address their own data quality deficiencies if they do *not* meet organizational standards while the data warehouse is used as an AFW knowledge source for enterprise data storage and mining of information that affects the entire AFW organization.

This investigation also shows the importance of developing an application prototype for performance benchmarking purposes, derived from a simple client/server model. The prototype is used to *quantify* the impact of design decisions on computer resource utilization, system scalability, system performance, ease of implementation, and system evolution (§4.2.2 CORBA Performance Metrics). These performance benchmarks are also used as *selection criteria* for *objec-*

tive purchasing decisions, whether its procuring the appropriate number of hardware components based on *measured* system efficiencies or determining an appropriate CORBA implementation that *meets* the minimum performance and functional requirements for the system.

7.2 Critical Research Factors

Four factors impacting this investigation are related to meeting the objectives set forth in chapter one (§1.4 Specific Objectives):

1. Functionality: Does the software system meet customer functional requirements? The TAF accuracy measurement model previously discussed provides an efficient, rule-based approach to the problem of processing and measuring TAF accuracy (§7.1 Research Significance Revisited).
2. Performance: Does ORB performance meet application requirements? Is the application tuned for its operating environment? A simple client/server measurement methodology is used to measure ORB performance. This model is then extended to application-specific functions and data types to quantify the impact of bulk operations, multithreading, and component granularity (§4.2.2 CORBA Performance Metrics, §6.2 ORB Experiments).
3. Complexity: Does the software design contain well-defined interfaces that hide details and complexities? Are sufficient software abstractions provided to simplify the architecture? The TAF accuracy measurement model is the principal abstraction upon which all further design is based (§3.4.1.5 Measur-

ing TAF Data Quality and Accuracy). The design methodology completely separates interface definition from implementation detail and uses OMG services to promote reliability, reusability, and architectural stability at the appropriate level (§3.5 Distributed Object System Design).

4. Scalability and Extensibility: Is the software design easily extended and scalable? Does the software system use a scalability model to solve design issues in the global, enterprise, system, application, and component-level architectural domains? The design is a scalable and easily extendable solution to a global processing problem where proven design patterns are applied and implemented at the appropriate architectural level (§3.5.3 System Performance, Evolution, and Reliability Considerations, §3.5.2 System Module Design, §6.5 Integration and System Experiments).

7.3 Efficiency and Effectiveness Discussion

The major research findings in the context of distributed system efficiency and methodology effectiveness are discussed in this section. First, the impact of a quantitative approach using system prototyping is presented then the overall effectiveness of the methodology is examined.

7.3.1 Impact of System Prototyping

All complex systems evolve from simple models. In the case of distributed object systems, the effectiveness of using a simple client/server model is used to show the impact of calling and sending specific data types to a remote machine. From this simple measurement model, the prototype is incrementally extended to include more and more distributed application functionality such as testing a vendor's CORBA implementation, a designer's algorithm selection, or a module's particular deployed configuration in a N-tier topology. The overall impact of using distributed system prototyping is that it provides valuable *insight* into *efficient* distributed application design. This information is used for software vendor selection, hardware purchases, or as empirical data that shows inefficient use of current hardware platforms as it relates to resource utilization and computing efficiency. The following factors require attention to enable efficient distributed design and component placement in a CORBA environment:

1. Call latency/dispatch rates for ORBs: Call dispatch rates for parameterless and parameterized operations set a fundamental design limit for an object request broker. If the TFMS required an operation dispatch rate of over 300 calls/second for a double data type, then other ORBs would have to be considered because MICO's dispatch rate was ~299 (§6.2 ORB Experiments).
2. Name Service and Event Service performance: Name resolution and binding times can greatly impact a large, distributed environment that relies on a Name Service to solve the problem of how components get object references at runtime. The MICO name service shows that the number of bindings in a naming context has no effect on client name resolution and binding (§6.2 ORB

Experiments). Many distributed environments are also event-driven, so it's important to stress the fan-out capability of the implementation. The MICO Event service is scalable for 1, 10, 100, and 1000 forecasts up to five collection sites, but when seven sites subscribed for publications, the time to publish 100 and 1000 forecasts increased dramatically. Test all CORBA services designed in the application's system architecture to determine if it meets system requirements (§6.3 Publication Experiments).

3. Data types and their corresponding return types: These results confirm the importance of designing remote interface operations with bulk data transfer support. The impact of providing bulk transfer support as operation parameters are shown to be an effective way to efficiently design CORBA IDL operations. A single forecast and a CORBA sequence of 10 forecasts incurred the same communication costs (§6.2 ORB Experiments).

4. Server/component computation times: Too little computation on the server/component reduces overall system efficiency because of the overhead incurred with distributed computing. The TFMS validation function exhibits good overall efficiency when validating 100 or 1000 forecasts, but system efficiency cannot be maintained when validating 1 or 10 forecasts (§6.4 Validation Experiments).

5. Task allocation and threads: Certain functions are more amenable to a multi-threaded approach. Allocate separate tasks for expensive I/O or computa-

tional processing. In the case of testing server performance, a single-threaded implementation is underutilized for a 10 KFLOP workload, but is fully utilized when a 1 MFLOP workload is applied. Overall server response time and throughput is improved by allocating a separate task (thread) for expensive workloads; this approach also provides hardware scalability when additional processors are added to the machine (§6.2 ORB Experiments).

6. Component Integration: Once the distributed characteristics of individual components are determined, integrate the components for system-wide transaction testing to determine end-to-end efficiency. When testing component integration, keep Amdahl's Law in mind: 1) make the common case fast, and 2) application speedup is bounded by the slowest component [10]. The submit transaction showed a dramatic decrease in execution time when 100 or 1000 forecasts are submitted to three validation modules, but levels off due to increased communication time when using five validation modules (§6.5 Integration and System Experiments).

7.3.2 Effectiveness of Methodology

A major focus of this research investigation was to describe an appropriate design methodology suitable for large distributed object systems (§III. Distributed System Design). Major findings include:

1. Interface and implementation separation is paramount to minimizing software dependencies and maximizing reuse in object-oriented systems (§3.5.2 System Module Design).
2. Software design patterns are a very effective form of guidance available for solving design and implementation issues at all architectural levels [13 and 14]. Design patterns are also very useful for designing easily extendable software systems. Use patterns at all architectural levels; e.g. use CORBA's Event Service as the publish/subscribe mechanism at the system level (§3.5.1 System Partitioning, §3.5.2 System Module Design).
3. Parameterized types aid system reuse and overall reliability. The C++ STL provides many different data structures, e.g. vector, list, stack, map, multimap, etc. [9]. Using the C++ STL increases application reliability and software reuse while reducing coding and debugging time since its containers and algorithms are fully tested and debugged [51 and 70].
4. Structured analysis techniques, e.g. data/control flow diagrams are more effective for properly partitioning the distributed system into modules and specifying the task architecture (§3.4.2 Information System Model).
5. OO modeling worked best to show the system's schema, message paths between system entities, and internal task behavior (§3.4.2.4 Transaction Model, §3.5 Distributed Object System Design).

The effectiveness of this approach is primarily found in the unification of many *proven* software analysis, design, implementation, and performance prediction techniques used to *efficiently and incrementally* develop a large, distributed object system. This methodology uses proven techniques at each stage of the lifecycle. By using the strengths of structured, modular, and object-oriented software design methods, a loosely coupled and highly autonomous design is produced. System extension and modification to component/module implementations has no effect on clients, since they're compiled to stable interfaces. By using proven design patterns to solve problems at each architectural level, an extensible, efficient, and understandable software system is developed. The use of parameterized types (generics) along with OO programming techniques reduces source code errors by using preexisting component/standard libraries and application classes, e.g. Threadpool application class and C++ map container, to implement either application or additional object-level functionality.

7.4 Future Research and Recommendations

The TFMS prototype provides simple event service and name service implementations. Future research efforts could explore federating the OMG Event, Name, or Trader services to investigate their use as scalable, global architectures. The OMG Event service enables site autonomy by completely decoupling clients from servers, as location independence is a major goal concerning open system design [12 and 21]. Future research efforts could also expand upon this notion by implementing *dynamic discovery* characteristics provided by the CORBA Query, Notification, and Trader services. Research in this area has direct application in knowledge or agent-based systems where component mobility and autonomy are the overriding system goal [8].

The basic TFMS N-tier application and test generator is built to perform a variety of tests or implement certain functionality: call latency, parameter marshaling, Win32 API threads, event service, and name service. Many of the experiments are useful in determining the best performing ORB for an organization's business requirements, or if a different distributed object paradigm, e.g. DCOM or Java RMI are more appropriate. Additional development could vastly improve the comprehensiveness of the measurement model for all distributed object paradigms with a more intuitive graphical user interface. This measurement model could address real-time distributed application models as well, with particular emphasis on priority inversion bounding and quality of service guarantees.

Appendix A: Object Management Architecture (OMA)

To support very large, complex distributed object applications, it's desirable to specify an infrastructure that supports the handling of common operations such as object lifecycles, identification, interface definitions, and intercommunication. The Object Management Group (OMG) was formed to reduce complexity and lower development cost and time. The OMG is an international trade organization incorporated as a nonprofit organization in the United States. OMG is currently comprised of over 800 corporate members and the number gets larger every year [21]. OMG provides the OMA, which consists of all the terms and definitions that all specifications are based [26]. The OMA contains the following elements as depicted in Figure 68:

1. Object Request Broker (ORB): A communication standard known commercially as CORBA. CORBA 2.0 specified the Internet Inter-ORB Protocol (IIOP), which guarantees ORB interoperability if the vendor's ORB is CORBA 2.0 compliant [21].
2. Object Services: Common object specifications such as naming, security, and transaction and are known commercially as Common Object Service Specifications (COSS). The COSS are collections of system-level services packaged as components specified in IDL [8].

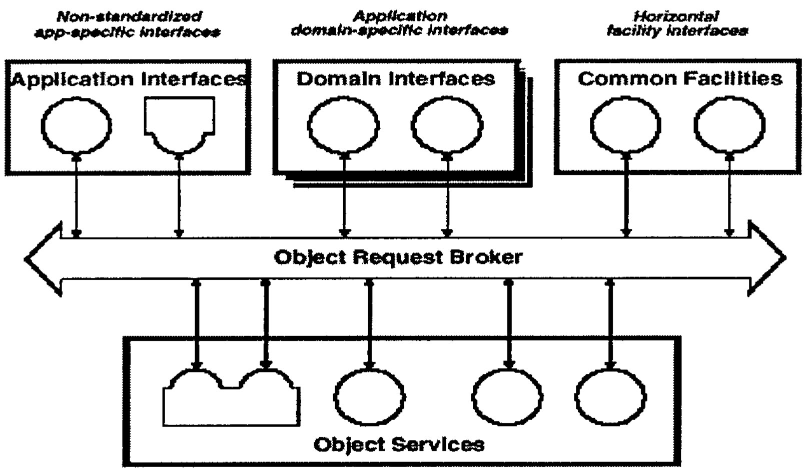

Non-standardized app-specific interfaces
Application domain-specific interfaces
Horizontal facility interfaces
Application Interfaces
Domain Interfaces
Common Facilities
Object Request Broker
Object Services

Figure 68: OMA [26]

3. Common Facilities: A set of *horizontal* and *vertical* IDL-specified services. Horizontal services may apply to more than one application domain such as information and system management while vertical services apply to a particular domain such as finance or healthcare.

4. Domain Interfaces: Specific application domains such as finance and healthcare [26].

5. Application Objects: Components specified for end-user applications. Application objects build on the services provides by the OMA.

The OMA is broken down into two main models: an Object Model and a Reference Model. The Object Model *defines* how the interfaces of objects distributed across a heterogeneous environment are described, and the Reference Model *characterizes* interactions between object interfaces [21].

A.1 The OMG Object Model

The Object Model defines an object as an encapsulated entity with an immutable distinct identity whose services are accessed only through a well-defined interface [26]. As the previous statement may show some readers, the OMA has some unique concepts and terminology associated with it. We explore a few concepts and terms regarding the OMA in this section.

A.1.1 General Concepts and Terminology

General terms and concepts related to CORBA [21]:

1. A *CORBA object* is a *virtual* entity capable of being located by an ORB and its operations invoked by a client. The "virtual" is regarding the fact that it doesn't exist unless it's made concrete by an implementation language such as C++ or Java.
2. A CORBA object servicing a client request is called a *target object*. The CORBA Object Model is *single dispatching*, where the target object is determined solely by an object reference.
3. A *client* is an entity that invokes a CORBA object. A *server* is an application where one or more CORBA objects exist. Of course, the term client and server are meaningful only if considering the request context.
4. A *request* invokes an operation on a CORBA object.
5. An *object reference* is used to identify, locate, and address a CORBA object. Object references are *opaque* to clients – only used for method invocation.

6. A *servant* is the CORBA object implementation in a particular programming language (class). Servants *incarnate* CORBA objects and can be considered object *instances* of a particular *class*.

A.1.2 OMG Interface Definition Language

Metadata is a crucial ingredient when developing flexible distributed systems. Metadata provides a distributed system with self-describing, dynamic, and reconfigurable capabilities. Using metadata, components discover each other at runtime, further enhancing interoperability [8]. Because IDL is a *declarative* language, its sole purpose is to allow object interfaces to be defined in a manner entirely independent of any particular programming language [21]. This allows applications implemented in different programming languages to interoperate; this language neutrality is critical to the OMA supporting heterogeneous environments [8, 21, and 26]. Language mappings specify how IDL is transformed into a particular programming language e.g. in C++, interfaces transform to classes and in Java, interfaces transform to public interfaces.

As Figure 69 shows, IDL *data types* include built-in simple types like short and string, and also constructed types such as enumeration, sequence and array. Object references are denoted in IDL just as many programming languages denote user-defined structures or classes, by using the name of the interface as the type. Multiple interface inheritance is supported in IDL as well.

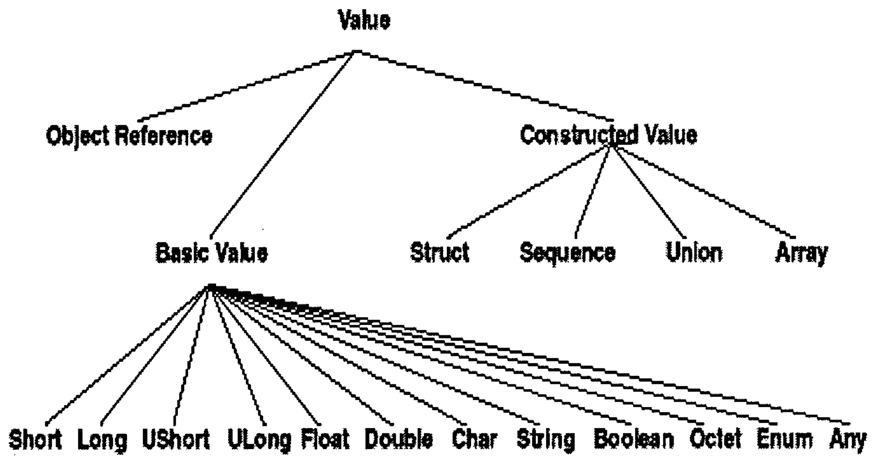


Figure 69: OMA Legal Values [26]

Modules provide a namespace to a group of *interface* definitions and are analogous to C++ namespaces and Java packages. *Interfaces* define a set of methods a client may invoke and map to C++ and Java as mentioned above. *Operations* denote a service (method) that clients may invoke. Operation *parameters* have mode in, out, or inout with respect to the servant. Parameter modes are necessary for two primary reasons:

1. Directional attributes are required for efficiency e.g. an "out" parameter is only communicated from the server to the client [12].
2. Directional attributes determine responsibility for memory management, whether the server or client is responsible to allocate memory for a specific parameter [21].

A.2 The OMG Reference Model

As mentioned above, the Reference Model provides interface categories that are general groupings for object interfaces that collaborate to carry out a set of responsibilities - commonly referred to as *frameworks*. In this section the ORB and the COSS frameworks are presented.

A.2.1 Object Request Broker (ORB)

CORBA defines the interface specification for an OMA-compliant ORB. Clients are not aware of the communication mechanisms employed in the ORB, how objects are activated, how objects are implemented, or where objects are located. The ORB is the application building foundation for distributed system design using the OMA. The ORB ensures interoperability between applications in both homogeneous and heterogeneous environments. The OMG Interface Definition Language (IDL) provides the "glue", connecting objects in a standard way by defining the interfaces to CORBA objects. The CORBA specification has the following elements as shown in Figure 70:

1. ORB Core: The CORBA runtime infrastructure. The ORB Core interface is not specified by CORBA, and is therefore vendor specific.
2. ORB Interface: The standard interface written in IDL and provided by a CORBA- compliant ORB.
3. IDL Stubs: Generated by the IDL compiler for each interface defined in IDL. Stubs hide the low-level networking details of object communication from the client, while presenting a high-level, object type-specific application programming interface (API) [33].

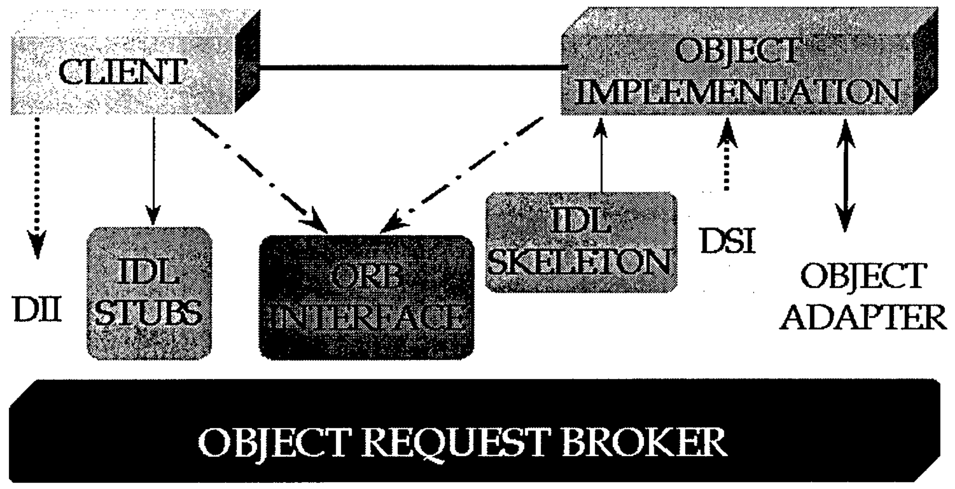


Figure 70: CORBA Specification

4. Dynamic Invocation Interface (DII): An alternative to static stubs for clients to "discover" and invoke objects. While static stubs provide an object type-specific API, DII provides a generic mechanism for constructing requests at run time. The interface repository, a client's object definition database of metadata, allows some measure of type checking to ensure that a target object can support the request made by the client.

5. Object Adapter: Provides extensibility of a CORBA-compliant ORB to integrate alternative object technologies into the OMA. For example, adapters may be developed to allow remote access to objects that are stored in an object-oriented database. Each CORBA-compliant ORB must support a specific object adapter called the basic object adapter (BOA). CORBA release 2.2 specifies the portable object adapter (POA), which removed server-side portability problems that existed in the BOA [21].

6. IDL Skeletons: The server-side analogue of IDL stubs. IDL skeletons receive requests for services from the object adapter, and call the appropriate operations in object implementations.
7. Dynamic Skeleton Interface (DSI): The server-side counterpart of DII. While IDL skeletons invoke specific operations in the object implementation, DSI defers this processing to the object implementation repository. This is useful for developing bridges and other mechanisms to support inter-ORB interoperation. The implementation repository is the server analogue to the interface repository; this is the server-side object definition database.

A.2.2 General Client/Sever Flow

Refer once again to Figure 70 for the following discussion. Requests flow down the client application, through the ORB, and up the server application as follows:

1. The client invokes a request into the ORB using either the static IDL stub or dynamic invocation interface (DII).
2. The client ORB transmits the request to the server ORB using IIOP.
3. The server ORB dispatches the request to the appropriate object adapter (BOA or POA).
4. The BOA or POA further dispatches the request to the appropriate servant object using either the static server skeleton or the dynamic skeleton interface.

5. The servant object that implements the interface definition performs the request and returns a response if required.

A.2.3 Application Development

To call a CORBA object member function, a client only needs to know the standard ORB Services and the object's IDL. Creating a CORBA Application involves the following generic steps:

1. Define object interfaces using the CORBA IDL.
2. Compile these interfaces with an IDL Compiler. This produces *stub code* for client objects and *skeleton code* for server objects.
3. Develop server programs that implement the IDL interfaces.
4. Register the server object(s) in the ORB.
5. Develop client programs that use the IDL interfaces.

A.2.4 Object Services

Object Services are domain-independent, horizontally oriented interfaces used by many end-user application programs. Major OMA object services:

1. Life Cycle: Used for creating, copying, moving and deleting components.
2. Persistence: Provides an interface for storing components persistently.
3. Name: Allows components to locate each other.
4. Event: Allows components to register and deregister for events.
5. Concurrency Control: Provides a resource lock manager.

6. Transactional: Provides two-phase commit using transactions.
7. Relationship: Allows the creation of dynamic relations among components.
8. Externalization: Provides a way of streaming data into or out of a component.
9. Query: Provides object query operations.
10. License: Controls object use.
11. Property: Provides a mechanism to alter component attributes.

Appendix B: CORBA IDL

```
/*-----------------------------------------------------------
Module: Terminal Forecast Management System (TFMS).
TYPE: CORBA Interface Definition Language (IDL)
DATE: 12/12/1999
FILENAME: TFMS.idl
DESCRIPTION: TFMS data and object interface definition.
AUTHOR: James S. Douglas, Captain, USAF
REVISIONS:
        12/12/1999: TFMS initial definition (Douglas - jsd)
-----------------------------------------------------------*/

// Uniqueness prefix applied for TFMS repository data.
#pragma prefix "douglas.com"

// Begin TFMS Namespace definition.
//-----------------------------------------------------------//
module TFMS
{
//*********************************************************************//

        // Begin TFMS data definitions
        //-----------------------------------------------------//
        enum MessageID
                {// scheduled, special, or forecasted weather report identifier
                METAR, SPECI, TAF};

        enum Modifier
                {// scheduled, ammended, corrected, or unscheduled
                REG, AMD, COR, RTD};

        struct DateTime
        {// TFMS Date/Time data structure
                unsigned short year;
                unsigned short quarter; // quarter = 1, 2, 3, or 4
                unsigned short month;
                unsigned short day;
                unsigned short hour;
                unsigned short minute;
        };

        struct ForecastData
        {// TFMS forecast data structure
                unsigned short icao;
                string<4> majcom;
                MessageID message_id;
                Modifier modifier;
                unsigned short wind_direction;
                unsigned short wind_speed;
                unsigned short wind_gusts;
                unsigned short visibility;
                string significant_weather;
                string cloud_layer;
                string remarks;
                string<14> key; // icao + year + month + day + hour
                DateTime issued;
                string change_group;
                unsigned short crosswind;
        };
        struct ObservationData
        {// TFMS observation data structure
                string<4> icao;
                string<4> majcom;
                MessageID message_id;
                Modifier modifier;
                unsigned short wind_direction;
                unsigned short wind_speed;
                unsigned short wind_gusts;
                unsigned short visibility;
                string significant_weather;
                string cloud_layer;
                string remarks;
                string<14> key; // icao + year + month + day + hour
                DateTime issued;
                unsigned short runway_visual_range;
        };

        /* Note: The use of struct types for forecasts and observations
        is to ensure compatibility across all object request brokers.  A
        more elegant definition would use objects by value, currently
        supported in CORBA version 2.3, but not         widely implemented.
        If using objects by value, a Weather interface is defined, then
        Forecast and Observation interfaces      would inherit all
        Weather attribute and operation definitions. (jsd) */

        // Defines sequences of forecast and observation data to
        // support bulk validation/publication requirements.
        typedef sequence<ForecastData> Forecasts;
        typedef sequence<ObservationData> Observations;

        struct ValidationField
        {// Contains validation information for a specific weather element.
    // The ValidationField data structure is used to collect metrics
    // for an individual weather element.
                MessageID type;
                string<14> key; // icao + year + month + day + hour
                string field;
                boolean passed_format;
                boolean passed_accuracy;
                string reason;
                string category;
                DateTime accuracy_id;
        };

        // Defines a sequence of validation fields as the
        // validation result.
        typedef sequence<ValidationField> ValidationFields;

        struct ValidationReport
        {// An individual validation report consists of a sequence of
         // validation fields.
                ValidationFields report;
        };

        // Defines a sequence of validation reports to
        // support bulk validation/metric processing requirements.
        typedef sequence<ValidationReport> ValidationReports;
        //---------------------------------------------------------//
        // End TFMS data definitions.

//*********************************************************************//
```

```
        // Begin TFMS object interface definitions
        //-------------------------------------------------------//
        interface IRBusiness
        {// IDL for remote Business interface

                // Submit operations.  All forecasts and observations
                // that pass validation are also published to other sites.
                // Pre: Requires a sequence of forecasts or observations.
                // Post: Returns a sequence of validation reports.
                // Note: TFMS Experiments do not require returning
                //                     validation results.
                void submitForecasts (in Forecasts tafs);
                ValidationReports submitObservations (in Observations obs);


                // Operation to manually validate forecasts.
                // Used for testing validation function.
                // Pre: Requires a sequence of forecasts.
                // Post: Returns a sequence of validation results.
                // Note: TFMS Experiments do not require returning
                //                     validation results.
                void validate (in Forecasts tafs);

                // Operation to manually publish forecasts.
                // Used for testing publication function.
                // Pre: Requires a sequence of forecasts and algorithm selection.
                // Primary copy = 1, Pipeline = 2, Event Channel = 3.
                // Post: None.
                void publish (in Forecasts tafs, in unsigned short algorithm);
        };

        interface IRCollection
        {// IDL for remote Collection interface.

                // Add operations.  All forecasts, observations,
                // and validation reports are processed for data collection.
                // Pre: Requires a sequence of forecasts, observations,
                // or validation reports.
                // Post: None.
                void addForecasts (in Forecasts tafs);
                void addObservations (in Observations obs);
                void addReports (in ValidationReports reps);
        };

        interface IRBenchmark
        {// IDL for ORB benchmark interface.

                oneway void callRate();
                void marshalDouble (in double d);
                void marshalForecast (in ForecastData fd);
                void marshalBulk (in Forecasts tafs);
                void singleThread (in unsigned long workload);
                void multiThread (in unsigned long workload);

        };
        //-------------------------------------------------------//
        // End TFMS object interface definitions

//**********************************************************************//
};
//---------------------------------------------------------//
// End Namespace definition for the Terminal Forecast Management System.
```

Acronyms

ACID	Atomicity, Consistency, Isolation, and Durability
AFI	Air Force Instruction
AFMAN	Air Force Manual
AFW	Air Force Weather
AFWA	Air Force Weather Agency
AWDS	Automated Weather Distribution System
C/S	Client/Server
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
COSS	Common Object Service Specifications
COTS	Commercial Off-The-Shelf
DBMS	Database Management System
DC	Domain Controller
DCOM	Distributed Component Object Model
DDBMS	Distributed Database Management System
DDL	Data Definition Language
DFD	Data Flow Diagram
DLL	Dynamic Link Library
DML	Data Manipulated Language
DOS	Distributed Object System

DTC	Distributed Transaction Coordinator
EXE	Executable
FAA	Federal Aviation Administration
FLOP	Floating-point Operations
FLOPS	Floating-point Operations per second
GOTS	Government Off-The-Shelf
ICAO	International Civil Aviation Organization
IAW	In Accordance With
IR	Information Retrieval
ISA	Instruction Set Architecture
KDC	Key Distribution Center
LAN	Local-Area Network
LPC	Local Procedure Call
MAN	Metropolitan-Area Network
MICO	MICO Is CORBA
MIPS	Millions of Instructions per second
MSMQ	Microsoft Message Queue Server
MTS	Microsoft Transaction Server
NOS	Network Operating System
NT	New Technology
ODBMS	Object-Oriented Database Management System

OMA	Object Management Architecture
OMG	Object Management Group
OO	Object Oriented
OOSE	Object Oriented Software Engineering
OQL	Object Query Language
ORB	Object Request Broker
OTM	Object Transaction Monitor
OWS	Operational Weather Squadron
PDC	Primary Domain Controller
POA	Portable Object Adapter
POS	Persistent Object Service
PPC	Pile of Personal Computers
RM	Resource Manager
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SMP	Symmetrical Multiprocessor
SQL	Structured Query Language
SRD	Systems Requirement Document
STL	C++ Standard Template Library
TAF	Terminal Aerodrome Forecast
TCP/IP	Transmission Control Protocol/Internet Protocol

TFMS	Terminal Forecast Management System
TGS	Ticket Granting Server
TPM	Transaction Processing Monitor
VB	Microsoft Visual Basic
VC	Microsoft Visual C++
WAN	Wide-Area Network
WF	Weather Flight
WMO	World Meteorological Organization
WWW	World Wide Web

Glossary

Terms and definitions related to this research:

Abstract Class: A class whose primary purpose is to define an interface. An abstract class cannot be instantiated within the software system.

Abstract Operation: An operation that declares a signature but doesn't implement the operation. In C++, this is a pure virtual member function.

Aggregate Object: An object composed of other objects.

Authentication: Provides a means to verify the identities of clients and servers.

Black box: A style of reuse based on class aggregation or a style of testing where internal implementation details are *not* used or revealed.

Class: A class defines an object's interface and implementation.

Client: An entity that invokes a CORBA object.

Concrete Class: A class with no abstract operations. A concrete class can be instantiated within the software system.

Concurrency: Although a transaction is an individual task within the system, multiple transactions may need to access the same data at the same time.

Constructor: An operation that defines the way to initialize an object within a particular class of objects.

CORBA object: A *virtual* entity capable of being located by an ORB and its operations invoked by a client. The "virtual" is regarding the fact that it doesn't exist unless it's made concrete by an implementation language such as C++.

Coupling: The degree of dependency between software components. Tight coupling means objects are highly dependent on each other while loose coupling usually refers to objects that have no dependencies on each other.

Data Allocation Schema: Describes where data fragments are located (site partitioning) [28].

Data Fragmentation Schema: Describes how global relations (data) are divided among local data stores.

Data replication: Signifies that multiple copies of data exist in the distributed system to improve fault tolerance and performance [9].

Delegation: An implementation method where an object delegates a request to another object who then carries out the request.

Design Pattern: Addresses recurring design problems in object systems. Usually describes the problem, circumstances for applying the pattern, structural examples, and consequences of using the design pattern.

Distributed Database: A *logical collection* of shared data, *physically distributed* across the nodes of a computer network [28].

Encapsulation: Result of hiding an object's implementation or internal state.

Inheritance: Consists of interface and implementation inheritance. In CORBA, *multiple* interface inheritance is supported to *derive* new interfaces from existing ones. Conventional class inheritance typically refers to the combination of interface and implementation inheritance. A class that inherits from another class is called a subclass or derived class.

Integrity: Refers to a transaction transforming the database from one consistent state to another consistent state.

Interface: Describes the set of operations or services an object provides to clients. In distributed object systems, you program to an interface, not an implementation.

Localized failure: Denotes that if a node in the distributed system fails, it doesn't affect the operation of other nodes [27 and 28].

Location Transparency: Signifies that users do not know or need to know where data is stored on the network [12].

Object Reference: Used to identify, locate, and address a CORBA object. Object references are *opaque* to clients - only used for method invocation.

Operation: An object's data is manipulated by operations. Operations are performed when the object receives a request.

Overriding: Redefining an inherited operation in a derived class.

Parameterized Type: A type that requires additional type specification supplied as parameters during declaration. Called *templates* in C++, *generics* in Ada.

Polymorphism: The ability to substitute objects with matching interfaces for one another during run time. In C++, a dynamic cast operation is used to downcast to the appropriate object type. CORBA uses a narrow operation to downcast.

Principal: A user or process that requires secure communication.

Recovery: Refers to the ability to *rollback* all intermediate changes if a particular action cannot be completed, so the database maintains integrity and is not left in an inconsistent (unknown) state.

Request: Invocation of an operation on a CORBA object.

Secret Key Cryptosystem: Uses a single key for both encryption and decryption.

Servant: A CORBA object implementation in a particular programming language. Servants *incarnate* CORBA objects and can be considered object *instances* of a particular *class*.

Server: An application where one or more CORBA objects exist.

Single Logical Database View: Analogous to the *single system image* notion where the distributed system appears to the user as a centralized or local system [10, 12, and 27].

Target Object: A CORBA object servicing a client request.

Traceability: Validation that a particular software function corresponds to a user-specified requirement.

Transaction: Considered a logical unit of work, recovery, integrity, and concurrency in a database system [9].

Type: The name of a particular interface or class.

White Box: A style of reuse based on class inheritance or a style of testing where internal implementation details are used or revealed.

Work: Refers to performing a required system action.

Bibliography

1. Department of the Air Force. System Requirements Document (SRD) For the Reengineered Air Force Weather Weapon System (AFWWS). Electronic Systems Center Air Force Weather Systems (ESC/ACW), 1998.
2. Valente, Alexandre. Analysis of the Use of N-Tier Architecture with Distributed Objects in Distributed-Database Systems. MS thesis, AFIT/GCE/ENG/99J. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, June 1998.
3. Semaphore Report. The Object Oriented Lifecycle: A Practical Guide to Best Practice Based Development of Object-Oriented Systems. Revision 980314, 1997.
4. Schlicher, Bob. "CORBA in the Enterprise." Excerpt from unpublished article, not paginated. http://developer.netscape.com, no date.
5. Rumbaugh, James and others. Object-Oriented Modeling and Design. Prentice-Hall, Inc., 1991.
6. Pressman, Roger. Software Engineering: A Practitioner's Approach, 4th ed. McGraw-Hill, 1997.
7. Edwards, Jeri. 3-Tier Client/Server at Work. John Wiley & Sons, 1998.
8. Orfali, Robert and others. The Essential Distributed Objects Survival Guide. John Wiley & Sons, Inc., 1996.
9. Date, Chris. An Introduction to Database Systems, 6th ed. Addison-Wesley, 1995.
10. Hwang, Kai and Zhiwei Xu. Scalable Parallel Computing. McGraw-Hill, 1998.
11. Silberschatz, Abraham and Peter Galvin. Operating System Concepts, 5th ed. Addison-Wesley, 1998.
12. Tanenbaum, Andrew. Distributed Operating Systems. Prentice-Hall, 1995.
13. Gamma, Erich and others. Design Patterns: Elements of Reusable Objected-Oriented Software. Addison-Wesley, 1995.
14. Mowbray, Thomas and Raphael Malveau. CORBA Design Patterns. John Wiley & Sons, Inc., 1997.
15. Kumar, Vipin and others. Introduction to Parallel Computing. The Benjamin/Cunnings Publishing Company, 1994.

16. Wilson, Chip. "Application Architectures with Enterprise JavaBeans," Component Strategies, 25-34 (August 1999).

17. Box, Don. Essential COM. Addison-Wesley, 1998.

18. Chung, P. E. and others. "DCOM and CORBA Side by Side, Step By Step, and Layer by Layer," C++ Report, (September 1997).

19. McConnel, Steve. Code Complete: A Practical Handbook of Software Construction. Microsoft Press, 1993.

20. Kim, Won. Modern Database Systems: The Object Model, Interoperability, and Beyond. Addison-Wesley, 1995.

21. Henning, Mike and Steve Vinoski. Advanced CORBA Programming with C++. Addison-Wesley, 1999.

22. Stroustrup, Bjarne. The C++ Programming Language, 3rd ed. Addison-Wesley, 1997.

23. Software Engineering Institute Technical Report. Distributed Object Technology with CORBA and Java: Key Concepts and Implications. CMU/SEI-97-TR-004, June 1997.

24. Tiwary, Ashutosh and others. "Building Large Distributed Software Systems Using Objects," OOPSLA '95, 191-195, (October 1995).

25. Arcus Report. Decoupling of Object-Oriented Systems: A Collection of Patterns. Revision 1.0, November 1996.

26. Object Management Group (OMG). A Discussion of the Object Management Architecture. OMG, January 1997.

27. Andleigh, Prabhat and Michael Gretzinger. Distributed Object-Oriented Data-Systems Design. Prentice-Hall, 1992.

28. Bell, David and Jane Grimson. Distributed Database Systems. Addison-Wesley, 1992.

29. Booch, Grady. Object-Oriented Design with Applications. Benjamin/Cummings, 1991.

30. Fujii, Roger. Logicon, Inc. "Introduction to Software Validation and Validation." Presented to Air Force Institute of Technology, Department of Electrical and Computer Engineering students and faculty. Air Force Institute of Technology, Wright-Patterson AFB OH, 5 August 1999.

31. Berard, Edward. Essays on Object-Oriented Software Engineering, Volume I. Prentice-Hall, 1993.

32. Farley, Jim. Java Distributed Computing. O'Reilly & Associates, 1998.

33. Software Engineering Institute Technical Report. Common Object Request Architecture. http://www.sei.cmu.edu/str/descriptions/corba.html, January 1997.

34. Romer, Kay and Arno Puder. MICO Is CORBA. http://www.mico.org. Version 2.2.7, no date.

35. Clark, Laurence. "Oracle Fail-Safe Solutions for Windows NT Clusters," Oracle Whitepaper, (September 1998).

36. Microsoft Developer Network. "Two-phase Commit Protocol," Microsoft Web Site, http://msdn.microsoft.com, no date.

37. National Weather Service. National Weather Service Operations Manual: Aviation Terminal Forecasts. WSOM Issuance 97-5. NWS, 6 June 1997.

38. Ozsu, M. Tamer Patrick Valduriez. "Distributed and Parallel Database Systems," ACM Computing Surveys, Vol. 28, No. 1, (March 1996).

39. Tjandra, I. A. "Modeling Distributed Transactions," PDPTA '98 International Conference, (1998).

40. Hu, Jian and others. "CORBA as Infrastructure for Database Interoperability," Proceeding of the 10th IASTED International Conference on Parallel and Distributed Systems, (October 1998).

41. Agarwal, Shailesh and Arthur Keller. "Architecting Object Applications for High Performance with Relational Databases," Persistence Software Whitepaper, http://www.persistence.com, no date.

42. Cahoon, Brendon and Kathryn McKinley. "Performance Evaluation of a Distributed Architecture for Information Retrieval," ACM SIGIR '96, (1996).

43. Raj, Rajendra. "The Active Collection Framework," ACM Applied Computing Review, Vol. 7, No. 1, (Spring 1999).

44. Jordan, David. C++ Object Databases. Addison-Wesley, 1998.

45. Jackson, Richard and others. "Guidelines for Reporting Results of Computational Experiments. Report of the Ad Hoc Committee," Mathematical Programming 49 (1991), 413-425, (Revised September 1990).

46. Leon, Darryl. Real-time Forecast Validation for the Reengineered Air Force Weather Architecture. Excerpt from unpublished MS thesis, AFIT/GCS/ENG/00M-14. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 2000.

47. Uhler, Stephen. "Event-Based Servers in Tcl," Dr. Dobb's Journal, 74-78 (September 1999).

48. Crowder, Harlan and others. "On Reporting Computational Experiments with Mathematical Software," ACM Transactions on Mathematical Software, Vol. 5, No., 2, 193-203, (June 1979).

49. Defense Information Systems Agency. Predicting CORBA Performance Through Prototyping. Joint Interoperability & Engineering Organization, Center for Computer Systems Engineering, no date.

50. Beveridge, Jim and Robert Wiener. Multithreading Applications in Win32: The Complete Guide to Threads. Addison-Wesley, 1997.

51. Soukup, Jiri. "Data Structures as Objects," Dr. Dobb's Journal, 21-30 (October 1999).

52. Lippman, Stanley. "Improving C++ Program Performance," Dr. Dobb's Journal, 40-45 (October 1999).

53. Microsoft Developer Network. "Validating the Data," Microsoft Web Site, http://msdn.microsoft.com, no date.

54. Gray, Jim. The Benchmark Handbook: For Database and Transaction Processing Systems, 2nd ed. Morgan Kaufmann, 1993.

55. Hennessy, John and David Patterson. Computer Organization and Design: The Hardware/Software Interface, 2nd ed. Morgan Kaufmann, 1998.

56. Schmidt, Douglas. "Evaluating Architectures for Multithreaded Object Request Brokers," Communications of the ACM, Vol. 41, No. 10, (October 1998).

57. Kramer, Jeff. "Distributed Software Engineering: Invited state-of-the-art Report," IEEE, (1994).

58. Object Management Group (OMG). The Common Object Request Broker: Architecture and Specification. OMG, February 1998.

59. Department of the Air Force. Meteorological Codes. AFM 15-124. Washington: HQ USAF/XOWP, 1 November 1998.

60. Schmidt, Douglas and Steve Vinoski. "Modeling Distributed Object Applications," C++ Report, (February 1995).

61. Spiegel, Murray. Probability and Statistics. McGraw-Hill, 1997.

62. Microsoft Developer Network. "Designing Efficient Applications for Microsoft SQL Server," Microsoft Web Site, http://msdn.microsoft.com, no date.

63. Thomas, Anne. Selecting Enterprise JavaBeans Technology. Patricia Seybold Group, July 1998.

64. Department of the Air Force. "Air Force Weather Metrics Program." PowerPoint Presentation, December 1998.

65. PeerLogic Whitepaper. DAIS Security. IN393 Issue 1, August 1998.

66. Object Management Group (OMG). CORBAServices: Common Object Services Specification: Security Service Specification. OMG, December 1998.

67. Department of the Air Force. Surface Observation Codes. AFM 15-111. Washington: HQ USAF/XOWP, 1 November 1998.

68. TAFVER IV Report. Functional/Statistical Requirements. No Revision, 1998.

69. Sumaria Systems Report. TAFVER IV Software Test Plan. Revision 1.0, 1998.

70. Musser, David and Atul Saini. STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library. Addison-Wesley, 1996.

71. Loshin, David. "Living Legacy," Intelligent Enterprise, 46-50 (December 21, 1999).

72. Monson-Haefel, Richard. "Validation Rules in Distributed Object Systems," Distributed Computing, 26-28 (November 1999).

73. Stillerman, Matthew and others. "Intrusion Detection for Distributed Applications," Communications of the ACM, Vol. 42, No. 7, (July 1999).

74. Kohl, John and others. "The Evolution of the Kerberos Authentication Service," IEEE, (1992).

75. Chappell, David. "Exploring Kerberos, the Protocol for Distributed Security in Windows 2000," Microsoft Systems Journal, (August 1999).

76. Gomaa, Hassan. Software Design Methods for Concurrent and Real-Time Systems. Addison-Wesley, 1993.

77. Cetus Links. Cetus Web Site, http://www.cetus-links.org, no date.

Vita

Captain James S. Douglas was born on 15 December 1965 in Butler, Pennsylvania. He graduated from Butler Area Senior High School in Butler, Pennsylvania in June 1983. Captain Douglas enlisted in the Air Force in 1984 and attained the rank of Technical Sergeant prior to completing undergraduate studies at the University of South Dakota in Brooking, South Dakota where he graduated with a Bachelor of Science degree in Electronic Engineering in May 1994. He was commissioned through Officer Training School at Maxwell AFB, Alabama where he was recognized as a Distinguished Graduate and the recipient of the Thomas Jefferson Award for highest academic honors. Captain Douglas was subsequently nominated for a Regular Commission in 1995.

While enlisted, Captain Douglas served as a heavy equipment operator in Civil Engineering and worked in the Air Force's air launch cruise missile program as a precision measurement calibration technician. His first assignment as a commissioned officer was at Travis AFB, California in December 1995 as a network engineer supporting HQ 15th Air Force. In August 1998, he entered the Graduate School of Engineering, Air Force Institute of Technology. Upon graduation, he will be assigned to Hill AFB, Utah as a software engineer.

Permanent Address:
127 Fontana Street
Butler, PA 16001

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE March 2000	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE DISTRIBUTED OBJECT SYSTEM ENGINEERING FOR TERMINAL AERODROME FORECAST VALIDATION AND METRICS PROCESSING	5. FUNDING NUMBERS
6. AUTHOR(S) James S. Douglas, Captain, USAF	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 P Street, Building 640 WPAFB OH 45433-7765	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/00M-07
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Mr. George Coleman, GM-14, Acting Director, Plans and Programs Directorate HQ AFWA/XP 106 Peacekeeper Drive Offutt AFB, NE 68113-4039 DSN 271-3585	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Dr. Gary B. Lamont, ENG, DSN: 785-3636, ext. 4718 COMM: (937) 255-3636 ext. 4718

12a. DISTRIBUTION AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Distributed object systems are a very complex intertwining of heterogeneous hardware, software, and operating systems coupled with communication networks of varying protocols and capacities. Distributed components offer improved performance through parallel processing, improved expansion and scalability opportunities through modularity, improved availability through replication, and improved resource sharing and interoperability through interconnection. This research provides a distributed system design methodology to validate terminal forecasts and gather metrics for the Air Force Weather Agency. Proven principles such as component reuse and architectural development are applied through the use of parameterized types and design patterns. A client/server measurement model is developed to show the impact of design decisions on computer resource utilization, system scalability, system performance, ease of implementation, and system evolution. An experimental Common Object Request Broker Architecture (CORBA) application is implemented to quantify the approach's effectiveness toward selecting an appropriate CORBA implementation and deploying the application in a distributed environment. While this research specifically uses CORBA for system development, the methodology presented is easily mapped onto any client/server architecture.

14. SUBJECT TERMS Software Engineering, Distributed Objects, Components, Scalability, Efficiency, Threads, Tasks, CORBA, Performance Benchmarks	15. NUMBER OF PAGES 198
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL